ParMETIS*

Parallel Graph Partitioning and Sparse Matrix Ordering Library

Version 3.1

George Karypis, Kirk Schloegel and Vipin Kumar

{karypis, kirk, kumar}@cs.umn.edu

University of Minnesota, Department of Computer Science and Engineering
Army HPC Research Center
Minneapolis, MN 55455

August 15, 2003

Contents

1 Introduction

PARMETIS is an MPI-based parallel library that implements a variety of algorithms for partitioning and repartitioning unstructured graphs and for computing fill-reducing orderings of sparse matrices. PARMETIS is particularly suited for parallel numerical simulations involving large unstructured meshes. In this type of computation, PARMETIS dramatically reduces the time spent in communication by computing mesh decompositions such that the numbers of interface elements are minimized.

The algorithms in PARMETIS are based on the multilevel partitioning and fill-reducing ordering algorithms that are implemented in the widely-used serial package METIS [5]. However, PARMETIS extends the functionality provided by METIS and includes routines that are especially suited for parallel computations and large-scale numerical simulations. In particular, PARMETIS provides the following functionality:

- Partition unstructured graphs and meshes.
- Repartition graphs that correspond to adaptively refined meshes.
- Partition graphs for multi-phase and multi-physics simulations.
- Improve the quality of existing partitionings.
- Compute fill-reducing orderings for sparse direct factorization.
- Construct the dual graphs of meshes

The rest of this manual is organized as follows. Section 2 briefly describes the differences between this version and the previous major release. Section 3 describes the various algorithms that are implemented in PARMETIS. Section 4 describes the format of the basic parameters that need to be supplied to the routines. Section 5 provides a detailed description of the calling sequences for the major routines in PARMETIS. Finally, Section 6 describes software and hardware requirements and provides contact information.

2 What is New in This Version

ParMETIS, Version 3.x contains a number of changes over the previous major release (Version 2.x). Theses changes include the following:

- The names and calling sequence of all the routines have changed due to expanded functionality that has been provided in this release. Table 1 shows how the names of the various routines map from version to version. Note that Version 3.0 is fully backwards compatible with all previous versions of ParMETIS. That is, the old API calls have been mapped to the new routines. However, the expanded functionality provided with this release is only available by using the new calling sequences.
- The four adaptive repartitioning routines: ParMETIS_RepartLDiffusion, ParMETIS_RepartGDiffusion, ParMETIS_RepartRemap, and ParMETIS_RepartMLRemap have been replaced by a (single) implementation of a unified repartitioning algorithm [15], ParMETIS_V3_AdaptiveRepart, that combines the best features of the previous routines.
- Multiple vertex weights/balance constraints are supported for most of the routines. This allows ParMETIS to be used to partition graphs for multi-phase and multi-physics simulations.
- In order to optimize partitionings for specific heterogeneous computing architectures, it is now possible to specify the target sub-domain weights for each of the sub-domains and for each balance constraint. This feature, for example, allows the user to compute a partitioning in which one of the sub-domains is twice the size of all of the others.
- The number of sub-domains has been de-coupled from the number of processors in both the static and the adaptive partitioning schemes. Hence, it is now possible to use the parallel partitioning and repartitioning algorithms to compute a k-way partitioning independent of the number of processors that are used. Note that Version 2.0 provided this functionality for the static partitioning schemes only.
- Routines are provided for both directly partitioning a finite element mesh, and for constructing the dual graph of a mesh in parallel. In version 3.1 these routines have been extended to support mixed element meshes.

Version 1.0	Version 2.0	Version 3.0
PARKMETIS	ParMETIS_PartKway	ParMETIS_V3_PartKway
PARGKMETIS	ParMETIS_PartGeomKway	ParMETIS_V3_PartGeomKway
PARGMETIS	ParMETIS_PartGeom	ParMETIS_V3_PartGeom
PARGRMETIS	Not available	Not available
PARRMETIS	ParMETIS_RefineKway	ParMETIS_V3_RefineKway
PARUAMETIS PARDAMETIS Not available Not available	ParMETIS_RepartLDiffusion ParMETIS_RepartGDiffusion ParMETIS_RepartRemap ParMETIS_RepartMLRemap	ParMETIS_V3_AdaptiveRepart
PAROMETIS	ParMETIS_NodeND	ParMETIS_V3_NodeND
Not available	Not available	ParMETIS_V3_PartMeshKway
Not available	Not available	ParMETIS_V3_Mesh2Dual

Table 1: The relationships between the names of the routines in the different versions of ParMETIS.

3 Algorithms Used in PARMETIS

PARMETIS provides a variety of routines that can be used to compute different types of partitionings and repartitionings as well as fill-reducing orderings. Figure 1 provides an overview of the functionality provided by PARMETIS as well as a guide to its use.

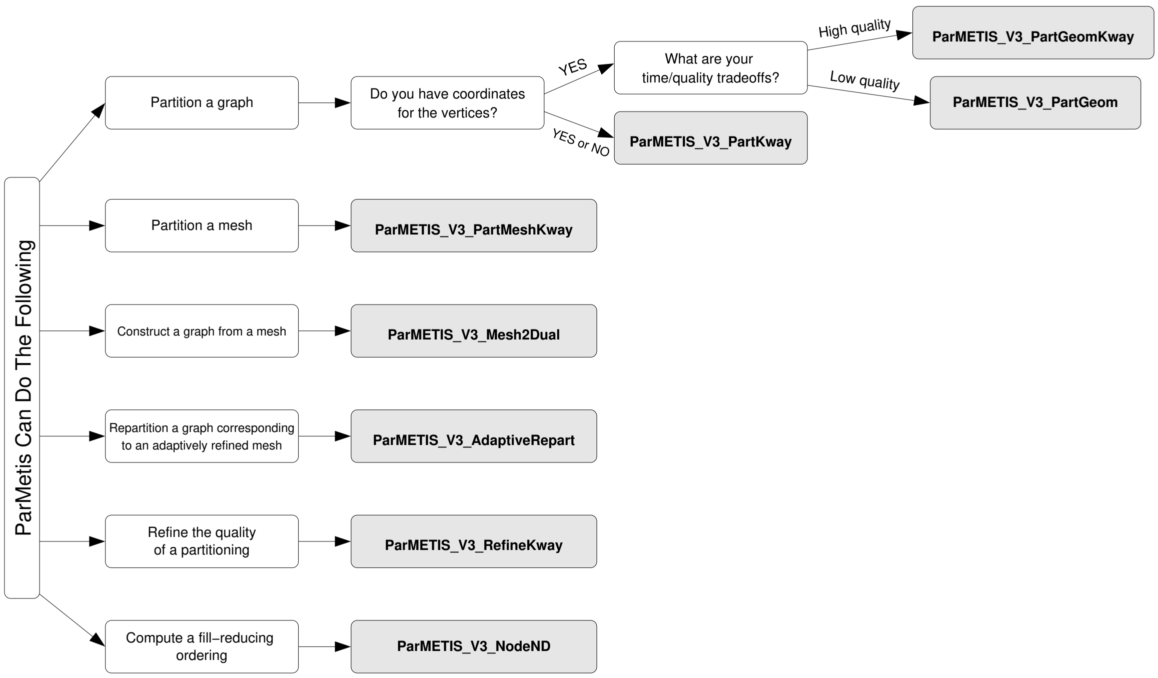


Figure 1: A brief overview of the functionality provided by PARMETIS. The shaded boxes correspond to the actual routines in PARMETIS that implement each particular operation.

3.1 Unstructured Graph Partitioning

ParMETIS_V3_PartKway is the routine in PARMETIS that is used to partition unstructured graphs. This routine takes a graph and computes a k-way partitioning (where k is equal to the number of sub-domains desired) while attempting to minimize the number of edges that are cut by the partitioning (*i.e.*, the *edge-cut*). ParMETIS_V3_PartKway makes no assumptions on how the graph is initially distributed among the processors. It can effectively partition a graph that is randomly distributed as well as a graph that is well distributed[1]. If the graph is initially well distributed among the processors, ParMETIS_V3_PartKway will take less time to run. However, the quality of the computed partitionings does not depend on the initial distribution.

The parallel graph partitioning algorithm used in ParMETIS_V3_PartKway is based on the serial multilevel k-way partitioning algorithm described in [6, 7] and parallelized in [4, 14]. This algorithm has been shown to quickly produce partitionings that are of very high quality. It consists of three phases: graph coarsening, initial partitioning, and uncoarsening/refinement. In the graph coarsening phase, a series of graphs is constructed by collapsing together

[1] The reader should note the difference between the terms *graph distribution* and *graph partition*. A partitioning is a mapping of the vertices to the processors that results in a distribution. In other words, a partitioning specifies a distribution. In order to partition a graph in parallel, an initial distribution of the nodes and edges of the graph among the processors is required. For example, consider a graph that corresponds to the dual of a finite-element mesh. This graph could initially be partitioned simply by mapping groups of n/p consecutively numbered elements to each processor where n is the number of elements and p is the number of processors. Of course, this naive approach is not likely to result in a very good distribution because elements that belong to a number of different regions of the mesh may get mapped to the same processor. (That is, each processor may get a number of small sub-domains as opposed to a single contiguous sub-domain). Hence, you would want to compute a new high-quality partitioning for the graph and then redistribute the mesh accordingly. Note that it may also be the case that the initial graph is well distributed, as when meshes are adaptively refined and repartitioned.

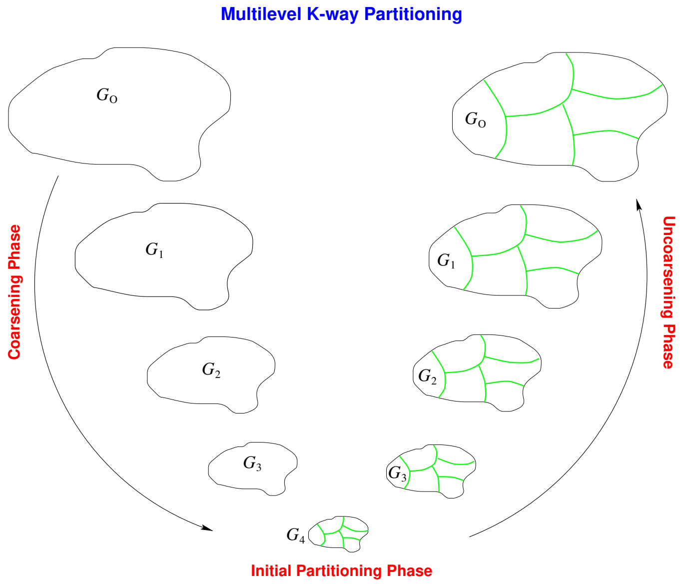


Figure 2: The three phases of multilevel k-way graph partitioning. During the coarsening phase, the size of the graph is successively decreased. During the initial partitioning phase, a k-way partitioning is computed, During the multilevel refinement (or uncoarsening) phase, the partitioning is successively refined as it is projected to the larger graphs. G_0 is the input graph, which is the finest graph. G_{i+1} is the next level coarser graph of G_i. G_4 is the coarsest graph.

adjacent vertices of the input graph in order to form a related coarser graph. Computation of the initial partitioning is performed on the coarsest (and hence smallest) of these graphs, and so is very fast. Finally, partition refinement is performed on each level graph, from the coarsest to the finest (*i.e.*, original graph) using a KL/FM-type refinement algorithm [2, 9]. Figure 2 illustrates the multilevel graph partitioning paradigm.

Comparisons performed in [7] have shown that serial multilevel k-way partitioning is over 50 times faster than multilevel spectral bisection [12] while producing partitionings that cut 10% to 50% fewer edges. Our experiments on a 1024-processor Cray T3E have shown that ParMETIS_V3_PartKway can partition a 500-million element 3D mesh in well under a minute!

Recall that we mentioned that if the graph is well distributed among the processors ParMETIS_V3_PartKway runs faster. In fact, our experiments have shown that when we use ParMETIS_V3_PartKway to partition a graph that is distributed according to the partitioning produced by an earlier call to ParMETIS_V3_PartKway, [2] the amount of time required for this second partitioning is often reduced by a factor of two to four. The reason for this has to do with the inter-processor communication pattern of ParMETIS_V3_PartKway. If the graph is initially distributed randomly (*i.e.*, there are many interface vertices), each processor spends a lot of time communicating information about these interface vertices to many other processors. On the other hand, if the graph is well distributed, the number of interface vertices is much smaller (as is the number of other processors with whom each processor has to communicate), reducing the overall runtime of the partitioner. Of course, this is the chicken and egg problem. How can we initially distribute the graph nicely without having first partitioned it? We have developed one such method for partitioning graphs that correspond to finite element meshes. This is to quickly compute a fairly good initial partitioning using the coordinate values of the mesh. We can then redistribute the graph according to this initial partitioning and then call ParMETIS_V3_PartKway on the redistributed graph. PARMETIS provides the ParMETIS_V3_PartGeomKway routine for doing just this. Given a graph that is distributed among the processors and the coordinates of the vertices,[3] ParMETIS_V3_PartGeomKway quickly computes an initial partitioning using a space-filling curve method, redistributes the graph according to this partitioning, and then calls ParMETIS_V3_PartKway to compute the final high-quality partitioning. Our experiments have shown that ParMETIS_V3_PartGeomKway is often two times faster

[2] That is, we first call ParMETIS_V3_PartKway to find a good partitioning of a graph. Next we move the vertices of the graph according to the computed partitioning, and then call ParMETIS_V3_PartKway to partition this same, but newly distributed, graph.

[3] ParMETIS_V3_PartGeomKway requires the coordinates of the centers of each element.

than ParMETIS_V3_PartKway, and achieves identical partition quality.

PARMETIS also provides the ParMETIS_PartGeom function for partitioning unstructured graphs when coordinates for the vertices are available. ParMETIS_PartGeom computes a partitioning based only on the space-filling curve method. Therefore, it is extremely fast (often 5 to 10 times faster than ParMETIS_PartGeomKway), but it computes poor quality partitionings (it may cut 2 to 10 times more edges than ParMETIS_PartGeomKway). This routine can be useful for certain computations in which the use of space-filling curves is the appropriate partitioning technique (*e.g.*, *n*-body computations).

3.2 Partitioning Meshes Directly

PARMETIS, Version 3.0 also provides new routines that support the computation of partitionings and repartitionings given *meshes* (and not graphs) as inputs. In particular, ParMETIS_V3_PartMeshKway take a mesh as input and computes a partitioning of the mesh elements. Internally, ParMETIS_V3_PartMeshKway uses a mesh-to-graph routine and then calls the same core partitioning routine that is used by both ParMETIS_V3_PartKway and ParMETIS_V3_PartGeomKway.

PARMETIS provides no such routines for computing adaptive repartitionings directly from meshes. However, it does provide the routine ParMETIS_V3_Mesh2Dual for constructing a dual graph given a mesh, quickly and in parallel. Since the construction of the dual graph is in parallel, it can be used to construct the input graph for ParMETIS_V3_AdaptiveRepart.

Essentially, both ParMETIS_V3_PartMeshKway and ParMETIS_V3_Mesh2Dual take the burden of writing an efficient mesh-to-graph routine from the user. Our experiments have shown that this routine typically runs in about half the time that it takes for PARMETIS to compute a partitioning.

3.3 Partitioning Adaptively Refined Meshes

For large-scale scientific simulations, the computational requirements of techniques relying on globally refined meshes become very high, especially as the complexity and size of the problems increase. By locally refining and de-refining the mesh either to capture flow-field phenomena of interest [1] or to account for variations in errors [11], adaptive methods make standard computational methods more cost effective. The efficient execution of such adaptive scientific simulations on parallel computers requires a periodic repartitioning of the underlying computational mesh. These repartitionings should minimize both the inter-processor communications incurred in the iterative mesh-based computation and the data redistribution costs required to balance the load. Hence, adaptive repartitioning is a multi-objective optimization problem. PARMETIS provides the routine ParMETIS_V3_AdaptiveRepart for repartitioning such adaptively refined meshes. This routine assumes that the mesh is well distributed among the processors, but that (due to mesh refinement and de-refinement) this distribution is poorly load balanced.

Repartitioning algorithms fall into two general categories. The first category balances the computation by incrementally diffusing load from those sub-domains that have more work to adjacent sub-domains that have less work. These schemes are referred to as ***diffusive schemes***. The second category balances the load by computing an entirely new partitioning, and then intelligently mapping the sub-domains of the new partitioning to the processors such that the redistribution cost is minimized. These schemes are generally referred to as ***remapping schemes***. Remapping schemes typically lead to repartitionings that have smaller edge-cuts, while diffusive schemes lead to repartitionings that incur smaller redistribution costs. However, since these results can vary significantly among different types of applications, it can be difficult to select the best repartitioning scheme for the job.

Recently, we developed a Unified Repartitioning Algorithm [15] for adaptive repartitioning that combines the best characteristics of remapping and diffusion-based repartitioning schemes. A key parameter used by this algorithm is the ***ITR Factor***. This parameter describes the ratio between the time required for performing the inter-processor communications incurred during parallel processing compared to the time to perform the data redistribution associated with balancing the load. As such, it allows us to compute a single metric that describes the quality of the repartitioning, even though adaptive repartitioning is a multi-objective optimization problem.

ParMETIS_V3_AdaptiveRepart is a parallel implementation of the Unified Repartitioning Algorithm. This is a multilevel partitioning algorithm, and so, is in nature similar to the the algorithm implemented in ParMETIS_V3_PartKway. However, this routine uses a technique known as *local coarsening*. Here, only vertices that have been distributed onto the same processor are coarsened together. On the coarsest graph, an initial partitioning need not be computed, as one can either be derived from the initial graph distribution (in the case when sub-domains are coupled to processors), or else one needs to be supplied as an input to the routine (in the case when sub-domains are de-coupled from processors). However, this partitioning does need to be balanced. The balancing phase is performed on the coarsest graph twice by alternative methods. That is, optimized variants of remapping and diffusion algorithms [16] are both used to compute new partitionings. A quality metric for each of these partitionings is then computed (using the ITR Factor) and the partitioning with the highest quality is selected. This technique tends to give very good points from which to start multilevel refinement, regardless of the type of repartitioning problem or the value of the ITR Factor. Note that the fact that the algorithm computes two initial partitionings does not impact its scalability as long as the size of the coarsest graph is suitably small [8]. Finally, multilevel refinement is performed on the balanced partitioning in order to further improve its quality. Since ParMETIS_V3_AdaptiveRepart starts from a graph that is already well distributed, it is extremely fast. Experiments on a 1024-processor Cray T3E show that ParMETIS_V3_AdaptiveRepart is able to compute partitionings for a billion-element mesh in under a minute.

Appropriate values to pass for the ITR Factor parameter can easily be determined depending on the times required to perform (i) all inter-processor communications that have occurred since the last repartitioning, and (ii) the data redistribution associated with the last repartitioning/load balancing phase. Simply divide the first time by the second. The result is the correct ITR Factor. In case these times cannot be ascertained (*e.g.*, for the first repartitioning/load balancing phase), our experiments have shown that values between 100 and 1000 work well for a variety of situations.

ParMETIS_V3_AdaptiveRepart can be used to load balance the mesh either before or after mesh adaptation. In the latter case, each processor first locally adapts its mesh, leading to different processors having different numbers of elements. ParMETIS_V3_AdaptiveRepart can then compute a partitioning in which the load is balanced. However, load balancing can also be done before adaptation if the degree of refinement for each element can be estimated *a priori*. That is, if we know ahead of time into how many new elements each old element will subdivide, we can use these estimations as the weights of the vertices for the graph that corresponds to the dual of the mesh. In this case, the mesh can be redistributed before adaption takes place. This technique can significantly reduce data redistribution times [10].

3.4 Partition Refinement

ParMETIS_V3_RefineKway is the routine provided by PARMETIS to improve the quality of an existing partitioning. Once a graph is partitioned and it has been redistributed accordingly, ParMETIS_V3_RefineKway can be called to compute a new partitioning that further improves the quality. Thus, like ParMETIS_V3_AdaptiveRepart, this routine assumes that the graph is already well distributed among the processors.

ParMETIS_V3_RefineKway can be used to improve the quality of partitionings that are produced by other partitioning algorithms (such as the technique discussed in Section 3.1 that is used in ParMETIS_V3_PartGeom). ParMETIS_V3_RefineKway can also be used repeatedly to further improve the quality of a partitioning. That is, we can call ParMETIS_V3_RefineKway, move the graph according to the partitioning, and then call ParMETIS_V3_RefineKway again. However, each successive call to ParMETIS_V3_RefineKway will tend to produce smaller improvements in quality.

Like ParMETIS_V3_AdaptiveRepart, ParMETIS_V3_RefineKway performs local coarsening. These two routines also use the same refinement algorithm. The difference is that ParMETIS_V3_RefineKway does not initially balance the partitioning on the coarsest graph, as ParMETIS_V3_AdaptiveRepart does. Instead, the assumption is that the graph is well distributed and the initial partitioning is balanced. Since ParMETIS_V3_RefineKway starts from a graph that is well distributed, it is very fast.

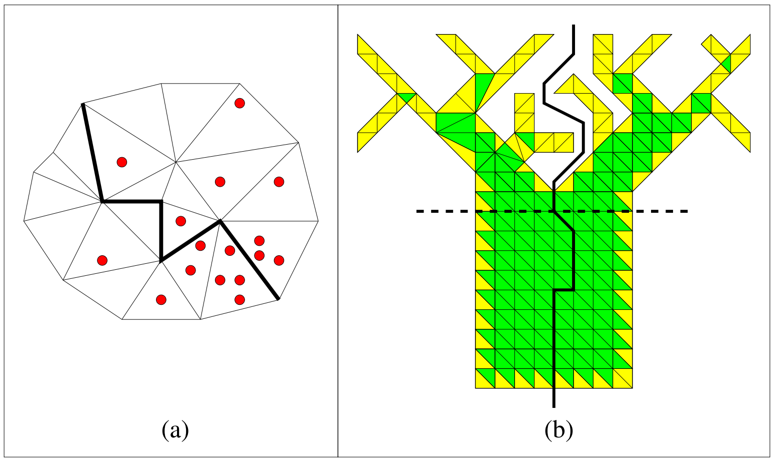


Figure 3: A computational mesh for a particle-in-cells simulation (a) and a computational mesh for a contact-impact simulation (b). The particle-in-cells mesh is partitioned so that both the number of mesh elements and the number of particles are balanced across the sub-domains. Two partitionings are shown for the contact-impact mesh. The dashed partitioning balances only the number of mesh elements. The solid partitioning balances both the number of mesh elements and the number of surface (lightly shaded) elements across the sub-domains.

3.5 Partitioning for Multi-phase and Multi-physics Computations

The traditional graph partitioning problem formulation is limited in the types of applications that it can effectively model because it specifies that only a single quantity be load balanced. Many important types of multi-phase and multi-physics computations require that multiple quantities be load balanced simultaneously. This is because synchronization steps exist between the different phases of the computations, and so, each phase must be individually load balanced. That is, it is not sufficient to simply sum up the relative times required for each phase and to compute a partitioning based on this sum. Doing so may lead to some processors having too much work during one phase of the computation (and so, these may still be working after other processors are idle), and not enough work during another. Instead, it is critical that every processor have an equal amount of work from each phase of the computation.

Two examples are particle-in-cells [17] and contact-impact simulations [3]. Figure 3 illustrates the characteristics of partitionings that are needed for these simulations. Figure 3(a) shows a mesh for a particles-in-cells computation. Assuming that a synchronization separates the mesh-based computation from the particle computation, a partitioning is required that balances both the number of mesh elements and the number of particles across the sub-domains. Figure 3(b) shows a mesh for a contact-impact simulation. During the contact detection phase, computation is performed only on the surface (i.e., lightly shaded) elements, while during the impact phase, computation is performed on all of the elements. Therefore, in order to ensure that both phases are load balanced, a partitioning must balance both the total number of mesh elements and the number of surface elements across the sub-domains. The solid partitioning in Figure 3(b) does this. The dashed partitioning is similar to what a traditional graph partitioner might compute. This partitioning balances only the total number of mesh elements. The surface elements are imbalanced by over 50%.

A new formulation of the graph partitioning problem is presented in [6] that is able to model the problem of balancing multiple computational phases simultaneously, while also minimizing the inter-processor communications. In this formulation, a weight vector of size m is assigned to each vertex of the graph. The *multi-constraint graph partitioning problem* then is to compute a partitioning such that the edge-cut is minimized and that every sub-domain has approximately the same amount of each of the vertex weights. The routines ParMETIS_V3_PartKway, ParMETIS_V3_PartGeomKway, ParMETIS_V3_RefineKway, and ParMETIS_V3_AdaptiveRepart are all able to compute partitionings that satisfy multiple balance constraints.

Figure 4 gives the dual graph for the particles-in-cells mesh shown in Figure 3. Each vertex has two weights here. The first represents the work associated with the mesh-based computation for the corresponding element. (These are all ones because we assume in this case that all of the elements have the same amount of mesh-based work associated with them.) The second weight represents the work associated with the particle-based computation. This value is estimated

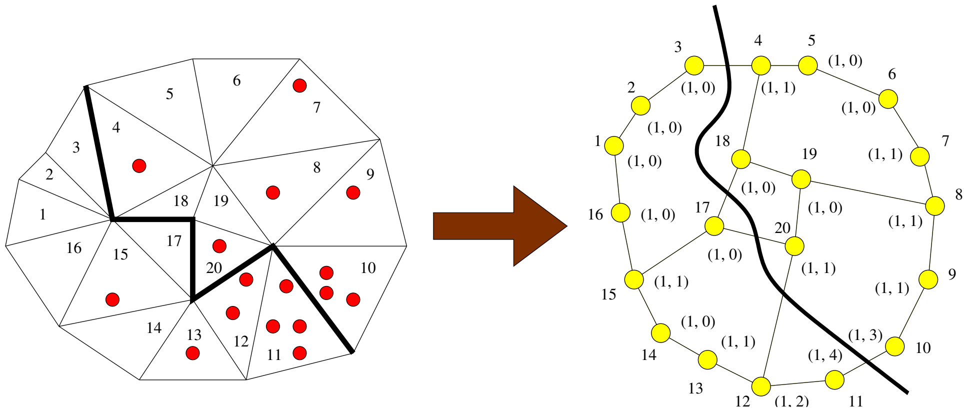


Figure 4: A dual graph with vertex weight vectors of size two is constructed from the particle-in-cells mesh from Figure 3. A multi-constraint partitioning has been computed for this graph, and this partitioning has been projected back to the mesh.

by the number of particles that fall within each element. A multi-constraint partitioning is shown that balances both of these weights.

A related graph partitioning problem formulation that is discussed in [13], allows edges to have multiple weights. We refer to this as the *multi-objective graph partitioning problem*. Multi-objective graph partitioning is applicable for tightly-coupled multi-physics computations involving multiple, spatially-overlapping meshes. Later versions of PARMETIS will support multiple edge weights.

3.6 Partitioning for Heterogeneous Computing Architectures

Complex, heterogeneous computing platforms, such as groups of tightly-coupled shared-memory nodes that are loosely connected via high bandwidth and high latency interconnection networks, and/or processing nodes that have complex memory hierarchies, are becoming more common, as they display competitive cost-to-performance ratios. The same is true of platforms that are geographically distributed. Most existing parallel simulation codes can easily be ported to a wide range of parallel architectures as they employ a standard messaging layer such as MPI. However, complex and heterogeneous architectures present new challenges to the scalable execution of such codes, since many of the basic parallel algorithm design assumptions are no longer valid. One way of alleviating this problem is to develop a new class of architecture-aware graph partitioning algorithms that optimally decompose computations given the architecture of the parallel platform. Ideally, this approach will alleviate the need for major restructuring of scientific codes.

We have taken the first steps toward developing architecture-aware graph-partitioning algorithms. These are able to compute partitionings that allow computations to achieve the highest levels of performance regardless of the computing platform. Specifically, we have enabled ParMETIS_V3_PartKway, ParMETIS_V3_PartGeomKway, ParMETIS_V3_PartMeshKway, ParMETIS_V3_RefineKway, and ParMETIS_V3_AdaptiveRepart to compute efficient partitionings for networks of heterogeneous processors. To do so, these routines require an additional array (`tpwgts`) to be passed as a parameter. This array describes the fraction of the total vertex weight each sub-domain should contain. For example, if you have a network of four processors, the first three of which are of equal processing speed, and the fourth of which is twice as fast as the others, the user would pass an array containing the values (0.2, 0.2, 0.2, 0.4). Note that by allowing users to specify target sub-domain weights as such, heterogeneous processing power can be taken into account when computing a partitioning. However, this does not allow us to take heterogeneous network bandwidths and latencies into account. Optimizing partitionings for heterogeneous networks is still the focus of ongoing research.

3.7 Computing Fill-Reducing Orderings

ParMETIS_V3_NodeND is the routine provided by PARMETIS for computing fill-reducing orderings, suited for

Cholesky-based direct factorization algorithms. ParMETIS_V3_NodeND makes no assumptions on how the graph is initially distributed among the processors. It can effectively compute fill-reducing orderings for graphs that are randomly distributed as well as graphs that are well distributed.

The algorithm implemented by ParMETIS_V3_NodeND is based on a multilevel nested dissection algorithm. This algorithm has been shown to produce low fill orderings for a wide variety of matrices. Furthermore, it leads to balanced elimination trees that are essential for parallel direct factorization. ParMETIS_V3_NodeND uses a multilevel node-based refinement algorithm that is particularly suited for directly refining the size of the separators. To achieve high performance, ParMETIS_V3_NodeND first uses ParMETIS_V3_PartKway to compute a high-quality partitioning and redistributes the graph accordingly. Next it proceeds to compute the $\log p$ levels of the elimination tree concurrently. When the graph has been separated into p parts (where p is the number of processors), the graph is redistributed among the processor so that each processor receives a single subgraph, and a multiple minimum degree algorithm is used to order these smaller subgraphs.

4 Input and Output Formats used by PARMETIS

4.1 Format of the Input Graph

All of the graph routines in PARMETIS take as input the adjacency structure of the graph, the weights of the vertices and edges (if any), and an array describing how the graph is distributed among the processors. Note that depending on the application this graph can represent different things. For example, when PARMETIS is used to compute fill-reducing orderings, the graph corresponds to the non-zero structure of the matrix (excluding the diagonal entries). In the case of finite element computations, the vertices of the graph can correspond to nodes (points) in the mesh while edges represent the connections between these nodes. Alternatively, the graph can correspond to the dual of the finite element mesh. In this case, each vertex corresponds to an element and two vertices are connected via an edge if the corresponding elements share an edge (in 2D) or a face (in 3D). Also, the graph can be similar to the dual, but be more or less connected. That is, instead of limiting edges to those elements that share a face, edges can connect any two elements that share even a single node. However the graph is constructed, it is usually undirected.[4] That is, for every pair of connected vertices v and u, it contains both edges (v, u) and (u, v).

In PARMETIS, the structure of the graph is represented by the compressed storage format (CSR), extended for the context of parallel distributed-memory computing. We will first describe the CSR format for serial graphs and then describe how it has been extended for storing graphs that are distributed among processors.

Serial CSR Format The CSR format is a widely-used scheme for storing sparse graphs. Here, the adjacency structure of a graph is represented by two arrays, `xadj` and `adjncy`. Weights on the vertices and edges (if any) are represented by using two additional arrays, `vwgt` and `adjwgt`. For example, consider a graph with n vertices and m edges. In the CSR format, this graph can be described using arrays of the following sizes:

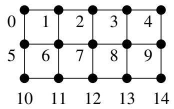
$$\texttt{xadj}[n+1],\ \texttt{vwgt}[n],\ \texttt{adjncy}[2m],\ \text{and}\ \texttt{adjwgt}[2m]$$

Note that the reason both `adjncy` and `adjwgt` are of size $2m$ is because every edge is listed twice (*i.e.*, as (v, u) and (u, v)). Also note that in the case in which the graph is unweighted (*i.e.*, all vertices and/or edges have the same weight), then either or both of the arrays `vwgt` and `adjwgt` can be set to `NULL`. ParMETIS_V3_AdaptiveRepart additionally requires a `vsize` array. This array is similar to the `vwgt` array, except that instead of describing the amount of work that is associated with each vertex, it describes the amount of memory that is associated with each vertex.

The adjacency structure of the graph is stored as follows. Assuming that vertex numbering starts from 0 (C style), the adjacency list of vertex i is stored in array `adjncy` starting at index $\texttt{xadj}[i]$ and ending at (but not including) index $\texttt{xadj}[i+1]$ (in other words, $\texttt{adjncy}[\texttt{xadj}[i]]$ up through and including $\texttt{adjncy}[\texttt{xadj}[i+1]-1]$). Hence, the adjacency lists for each vertex are stored consecutively in the array `adjncy`. The array `xadj` is used to point to where the list for each specific vertex begins and ends. Figure 5(b) illustrates the CSR format for the 15-vertex graph shown in Figure 5(a). If the graph was weights on the vertices, then $\texttt{vwgt}[i]$ is used to store the weight of vertex i. Similarly, if the graph has weights on the edges, then the weight of edge $\texttt{adjncy}[j]$ is stored in $\texttt{adjwgt}[j]$. This is the same format that is used by the (serial) METIS library routines.

Distributed CSR Format PARMETIS uses an extension of the CSR format that allows the vertices of the graph and their adjacency lists to be distributed among the processors. In particular, PARMETIS assumes that each processor P_i stores n_i consecutive vertices of the graph and the corresponding m_i edges, so that $n = \sum_i n_i$, and $2 * m = \sum_i m_i$. Here, each processor stores its local part of the graph in the four arrays $\texttt{xadj}[n_i+1]$, $\texttt{vwgt}[n_i]$, $\texttt{adjncy}[m_i]$, and $\texttt{adjwgt}[m_i]$, using the CSR storage scheme. Again, if the graph is unweighted, the arrays `vwgt` and `adjwgt` can be set to `NULL`. The straightforward way to distribute the graph for PARMETIS is to take n/p consecutive adjacency lists from `adjncy` and store them on consecutive processors (where p is the number of processors). In addition, each

[4] Multi-constraint and multi-objective graph partitioning formulations [6, 13] can get around this requirement for some applications. These also allow the computation of partitionings for bipartite graphs, as well as for graphs corresponding to non-square and non-symmetric matrices.



(a) A sample graph

Description of the graph on a serial computer (serial MeTiS)

xadj	0 2 5 8 11 13 16 20 24 28 31 33 36 39 42 44
adjncy	1 5 0 2 6 1 3 7 2 4 8 3 9 0 6 10 1 5 7 11 2 6 8 12 3 7 9 13 4 8 14 5 11 6 10 12 7 11 13 8 12 14 9 13

(b) Serial CSR format

Description of the graph on a parallel computer with 3 processors (ParMeTiS)

Processor 0:	xadj	0 2 5 8 11 13
	adjncy	1 5 0 2 6 1 3 7 2 4 8 3 9
	vtxdist	0 5 10 15
Processor 1:	xadj	0 3 7 11 15 18
	adjncy	0 6 10 1 5 7 11 2 6 8 12 3 7 9 13 4 8 14
	vtxdist	0 5 10 15
Processor 2:	xadj	0 2 5 8 11 13
	adjncy	5 11 6 10 12 7 11 13 8 12 14 9 13
	vtxdist	0 5 10 15

(c) Distributed CSR format

Figure 5: An example of the parameters passed to PARMETIS in a three processor case. The arrays `vwgt` and `adjwgt` are assumed to be NULL.

processor needs its local `xadj` array to point to where each of its local vertices' adjacency lists begin and end. Thus, if we take all the local `adjncy` arrays and concatenate them, we will get exactly the same `adjncy` array that is used in the serial CSR. However, concatenating the local `xadj` arrays will not give us the serial `xadj` array. This is because the entries in each local `xadj` must point to their local `adjncy` array, and so, `xadj[0]` is zero for all processors. In addition to these four arrays, each processor also requires the array `vtxdist`$[p+1]$ that indicates the range of vertices that are local to each processor. In particular, processor P_i stores the vertices from `vtxdist`$[i]$ up to (but not including) vertex `vtxdist`$[i+1]$.

Figure 5(c) illustrates the distributed CSR format by an example on a three-processor system. The 15-vertex graph in Figure 5(a) is distributed among the processors so that each processor gets 5 vertices and their corresponding adjacency lists. That is, Processor Zero gets vertices 0 through 4, Processor One gets vertices 5 through 9, and Processor Two gets vertices 10 through 14. This figure shows the `xadj`, `adjncy`, and `vtxdist` arrays for each processor. Note that the `vtxdist` array will always be identical for every processor.

All five arrays that describe the distributed CSR format are defined in PARMETIS to be of type `idxtype`. By default `idxtype` is set to be equivalent to type `int` (*i.e.*, integers). However, `idxtype` can be made to be equivalent to a `short int` for certain architectures that use 64-bit integers by default. (Note that doing so will cut the memory usage and communication time required approximately in half.) The conversion of `idxtype` from `int` to `short` can be done by modifying the file `parmetis.h`. (Instructions are included there.) The same `idxtype` is used for the arrays that store the computed partitioning and permutation vectors.

When multiple vertex weights are used for multi-constraint partitioning, the c vertex weights for each vertex are stored contiguously in the `vwgt` array. In this case, the `vwgt` array is of size nc, where n is the number of locally-stored vertices and c is the number of vertex weights (and also the number of balance constraints).

4.2 Format of Vertex Coordinates

As discussed in Section 3.1, PARMETIS provides routines that use the coordinate information of the vertices to quickly pre-distribute the graph, and so, speedup the execution of the parallel k-way partitioning. These coordinates are specified in an array called `xyz` of single precision floating point numbers (*i.e.*, `float`). If d is the number of dimensions of the mesh (*i.e.*, $d = 2$ for 2D meshes or $d = 3$ for 3D meshes), then each processor requires an array of size $d * n_i$, where n_i is the number of locally-stored vertices. (Note that the number of dimensions of the mesh, d, is required as a parameter to the routine.) In this array, the coordinates of vertex i are stored starting at location `xyz`[$i * d$] up to (but not including) location `xyz`[$i * d + d$]. For example, if $d = 3$, then the x, y, and z coordinates of vertex i are stored at `xyz[3*i]`, `xyz[3*i+1]`, and `xyz[3*i+2]`, respectively.

4.3 Format of the Input Mesh

The routine ParMETIS_V3_PartMeshKway takes a distributed mesh and computes its partitioning, while ParMETIS_V3_Mesh2Dual 2dual takes a distributed mesh and constructs a distributed dual graph. Both of these routines require an `elmdist` array that specifies the distribution of the mesh elements, but that is otherwise identical to the `vtxdist` array. They also require a pair of arrays called `eptr` and `eind`, as well as the integer parameter `ncommonnodes`.

The `eptr` and `eind` arrays are similar in nature to the `xadj` and `adjncy` arrays used to specify the adjacency list of a graph but now for each element they specify the set of nodes that make up each element. Specifically, the set of nodes that belong to element i is stored in array `eind` starting at index `eptr`[i] and ending at (but not including) index `eptr`[$i + 1$] (in other words, `eind[eptr`[i]`]` up through and including `eind[eptr`[$i + 1$]`-1]`). Hence, the node lists for each element are stored consecutively in the array `eind`. This format allows the specification of meshes that contain elements of mixed type.

The `ncommonnodes` parameter specifies the degree of connectivity that is desired between the vertices of the dual graph. Specifically, an edge is placed between two vertices if their corresponding mesh elements share at least g nodes, where g is the `ncommonnodes` parameter. Hence, this parameter can be set to result in a traditional dual graph (*e.g.*, a value of two for a triangle mesh or a value of four for a hexahedral mesh). However, it can also be set higher or lower for increased or decreased connectivity. ParMETIS_V3_PartMeshKway additionally requires an `elmwgt` array that is analogous to the `vwgt` array.

4.4 Format of the Computed Partitionings and Orderings

Format of the Partitioning Array The partitioning and repartitioning routines require that arrays (called `part`) of sizes n_i (where n_i is the number of local vertices) be passed as parameters to each processor. Upon completion of the PARMETIS routine, for each vertex j, the sub-domain number (*i.e.*, the processor label) to which this vertex belongs will have been written to `part`[j]. Note that PARMETIS does not redistribute the graph according to the new partitioning, it simply computes the partitioning and writes it to the `part` array.

Additionally, whenever the number of sub-domains does not equal the number of processors that are used to compute a repartitioning, ParMETIS_V3_AdaptiveRepart requires that the previously computed partitioning be passed as a parameter via the `part` array. (This is also required whenever the user chooses to de-couple the sub-domains from the processors. See discussion in Section 5.2.) This is because the initial partitioning needs to be obtained from the values supplied in the `part` array. If the numbers of sub-domains and processors are equal, then the initial partitioning can be obtained from the initial graph distribution, and so this information need not be supplied. (In this case, for each processor i, every element of `part` would be set to i.)

Format of the Permutation Array Likewise, each processor running ParMETIS_V3_NodeND writes its portion of the computed fill-reducing ordering to an array called `order`. Similar to the `part` array, the size of `order` is equal to the number of vertices stored at each processor. Upon completion, for each vertex j, `order`[j] stores the new global number of this vertex in the fill-reducing permutation.

Besides the ordering vector, ParMETIS_V3_NodeND also returns information about the sizes of the different

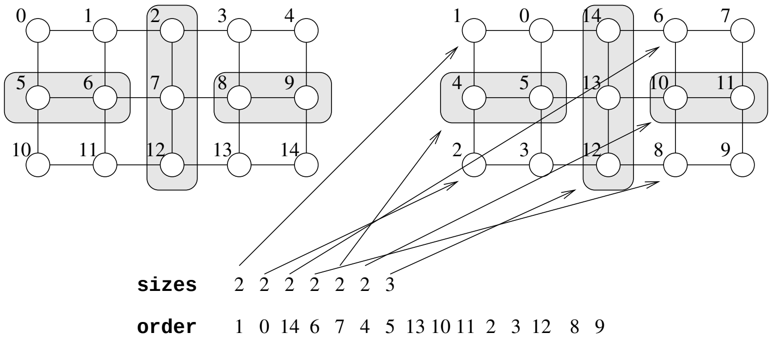


Figure 6: An example of the ordering produced by ParMETIS_V3_NodeND. Consider the simple 3×5 grid and assume that we have four processors. ParMETIS_V3_NodeND finds the three separators that are shaded. It first finds the big separator and then for each of the two sub-domains it finds the smaller. At the end of the ordering, the `order` vector concatenated over all the processors will be the one shown. Similarly, the `sizes` arrays will all be identical to the one shown, corresponding to the regions pointed to by the arrows.

sub-domains as well as the separators at different levels. This array is called `sizes` and is of size $2p$ (where p is the number of processors). Every processor must supply this array and upon return, each of the `sizes` arrays are identical.

The format of this array is as follows. The first p entries of `sizes` starting from 0 to $p-1$ store the number of nodes in each one of the p sub-domains. The remaining $p-1$ entries of this array starting from `sizes[`p`]` up to `sizes[`$2p-2$`]` store the sizes of the separators at the $\log p$ levels of nested dissection. In particular, `sizes[`$2p-2$`]` stores the size of the top level separator, `sizes[`$2p-4$`]` and `sizes[`$2p-3$`]` store the sizes of the two separators at the second level (from left to right). Similarly, `sizes[`$2p-8$`]` through `sizes[`$2p-5$`]` store the sizes of the four separators of the third level (from left to right), and so on. This array can be used to quickly construct the separator tree (a form of an elimination tree) for direct factorization. Given this separator tree and the sizes of the sub-domains, the nodes in the ordering produced by ParMETIS_V3_NodeND are numbered in a postorder fashion. Figure 6 illustrates the `sizes` array and the postorder ordering.

4.5 Numbering and Memory Allocation

PARMETIS allows the user to specify a graph whose numbering starts either at 0 (C style) or at 1 (Fortran style). Of course, PARMETIS requires that same numbering scheme be used consistently for all the arrays passed to it, and it writes to the `part` and `order` arrays similarly.

PARMETIS allocates all the memory that it requires dynamically. This has the advantage that the user does not have to provide workspace. However, if there is not enough memory on the machine, the routines in PARMETIS will abort. Note that the routines in PARMETIS do not modify the arrays that store the graph (*e.g.*, `xadj` and `adjncy`). They only modify the `part` and `order` arrays.

5 Calling Sequence of the Routines in ParMETIS

The calling sequences of the ParMETIS routines are described in this section.

5.1 Graph Partitioning

ParMETIS_V3_PartKway (idxtype *vtxdist, idxtype *xadj, idxtype *adjncy, idxtype *vwgt, idxtype *adjwgt, int *wgtflag, int *numflag, int *ncon, int *nparts, float *tpwgts, float *ubvec, int *options, int *edgecut, idxtype *part, MPI_Comm *comm)

Description

This routine is used to compute a k-way partitioning of a graph on p processors using the multilevel k-way multi-constraint partitioning algorithm.

Parameters

vtxdist This array describes how the vertices of the graph are distributed among the processors. (See discussion in Section 4.1). Its contents are identical for every processor.

xadj, adjncy
These store the (local) adjacency structure of the graph at each processor. (See discussion in Section 4.1).

vwgt, adjwgt
These store the weights of the vertices and edges. (See discussion in Section 4.1).

wgtflag This is used to indicate if the graph is weighted. *wgtflag* can take one of four values:

0 No weights (vwgt and adjwgt are both NULL).

1 Weights on the edges only (vwgt is NULL).

2 Weights on the vertices only (adjwgt is NULL).

3 Weights on both the vertices and edges.

numflag This is used to indicate the numbering scheme that is used for the *vtxdist*, *xadj*, *adjncy*, and *part* arrays. *numflag* can take one of two values:

0 C-style numbering that starts from 0.

1 Fortran-style numbering that starts from 1.

ncon This is used to specify the number of weights that each vertex has. It is also the number of balance constraints that must be satisfied.

nparts This is used to specify the number of sub-domains that are desired. Note that the number of sub-domains is independent of the number of processors that call this routine.

tpwgts An array of size `ncon` x `nparts` that is used to specify the fraction of vertex weight that should be distributed to each sub-domain for each balance constraint. If all of the sub-domains are to be of the same size for every vertex weight, then each of the `ncon` x `nparts` elements should be set to a value of 1/`nparts`. If `ncon` is greater than one, the target sub-domain weights for each sub-domain are stored contiguously (similar to the `vwgt` array). Note that the sum of all of the `tpwgts` for a give vertex weight should be one.

ubvec An array of size `ncon` that is used to specify the imbalance tolerance for each vertex weight, with 1 being perfect balance and `nparts` being perfect imbalance. A value of 1.05 for each of the `ncon` weights is recommended.

options This is an array of integers that is used to pass additional parameters for the routine. If *options[0]=0*, then the default values are used. If *options[0]=1*, then the remaining two elements of *options* are interpreted as follows:

options[1] This specifies the level of information to be returned during the execution of the algorithm. Timing information can be obtained by setting this to 1. Additional options for this parameter can be obtained by looking at the the file `defs.h` in the `ParMETIS-Lib` directory. The numerical values there should be added to obtain the correct value. The default value is 0.

options[2] This is the random number seed for the routine. The default value is 15.

edgecut Upon successful completion, the number of edges that are cut by the partitioning is written to this parameter.

part This is an array of size equal to the number of locally-stored vertices. Upon successful completion the partition vector of the locally-stored vertices is written to this array. (See discussion in Section 4.4).

comm This is a pointer to the MPI communicator of the processes that call PARMETIS. For most programs this will point to `MPI_COMM_WORLD`.

ParMETIS_V3_PartGeomKway (idxtype *vtxdist, idxtype *xadj, idxtype *adjncy, idxtype *vwgt, idxtype *adjwgt, int *wgtflag, int *numflag, int *ndims, float *xyz, int *ncon, int *nparts, float *tpwgts, float *ubvec, int *options, int *edgecut, idxtype *part, MPI_Comm *comm)

Description

This routine is used to compute a k-way partitioning of a graph on p processors by combining the coordinate-based and multi-constraint k-way partitioning schemes.

Parameters

vtxdist This array describes how the vertices of the graph are distributed among the processors. (See discussion in Section 4.1). Its contents are identical for every processor.

xadj, adjncy
These store the (local) adjacency structure of the graph at each processor. (See discussion in Section 4.1).

vwgt, adjwgt
These store the weights of the vertices and edges. (See discussion in Section 4.1).

wgtflag This is used to indicate if the graph is weighted. *wgtflag* can take one of four values:

- 0 No weights (vwgt and adjwgt are both NULL).
- 1 Weights on the edges only (vwgt is NULL).
- 2 Weights on the vertices only (adjwgt is NULL).
- 3 Weights on both the vertices and edges.

numflag This is used to indicate the numbering scheme that is used for the *vtxdist*, *xadj*, *adjncy*, and *part* arrays. *numflag* can take one of two values:

- 0 C-style numbering that starts from 0.
- 1 Fortran-style numbering that starts from 1.

ndims The number of dimensions of the space in which the graph is embedded.

xyz The array storing the coordinates of the vertices (described in Section 4.2).

ncon This is used to specify the number of weights that each vertex has. It is also the number of balance constraints that must be satisfied.

nparts This is used to specify the number of sub-domains that are desired. Note that the number of sub-domains is independent of the number of processors that call this routine.

tpwgts An array of size `ncon` x `nparts` that is used to specify the fraction of vertex weight that should be distributed to each sub-domain for each balance constraint. If all of the sub-domains are to be of the same size for every vertex weight, then each of the `ncon` x `nparts` elements should be set to a value of 1/`nparts`. If `ncon` is greater than one, the target sub-domain weights for each sub-domain are stored contiguously (similar to the `vwgt` array). Note that the sum of all of the `tpwgts` for a give vertex weight should be one.

ubvec An array of size `ncon` that is used to specify the imbalance tolerance for each vertex weight, with 1 being perfect balance and `nparts` being perfect imbalance. A value of 1.05 for each of the `ncon` weights is recommended.

options This is an array of integers that is used to pass parameters to the routine. Their meanings are identical to those of ParMETIS_V3_PartKway.

edgecut Upon successful completion, the number of edges that are cut by the partitioning is written to this parameter.

part This is an array of size equal to the number of locally-stored vertices. Upon successful completion the partition vector of the locally-stored vertices is written to this array. (See discussion in Section 4.4).

comm This is a pointer to the MPI communicator of the processes that call ParMETIS. For most programs this will point to `MPI_COMM_WORLD`.

Note

The quality of the partitionings computed by ParMETIS_V3_PartGeomKway are comparable to those produced by ParMETIS_V3_PartKway. However, the run time of the routine may be up to twice as fast.

ParMETIS_V3_PartGeom (idxtype *vtxdist, int *ndims, float *xyz, idxtype *part, MPI_Comm *comm)

Description

This routine is used to compute a p-way partitioning of a graph on p processors using a coordinate-based space-filling curves method.

Parameters

vtxdist This array describes how the vertices of the graph are distributed among the processors. (See discussion in Section 4.1). Its contents are identical for every processor.

ndims The number of dimensions of the space in which the graph is embedded.

xyz The array storing the coordinates of the vertices (described in Section 4.2).

part This is an array of size equal to the number of locally stored vertices. Upon successful completion stores the partition vector of the locally stored graph (described in Section 4.4).

comm This is a pointer to the MPI communicator of the processes that call PARMETIS. For most programs this will point to `MPI_COMM_WORLD`.

Note

The quality of the partitionings computed by ParMETIS_V3_PartGeom are significantly worse than those produced by ParMETIS_V3_PartKway and ParMETIS_V3_PartGeomKway.

ParMETIS_V3_PartMeshKway (idxtype *elmdist, idxtype *eptr, idxtype *eind, idxtype *elmwgt, int *wgtflag, int *numflag, int *ncon, int *ncommonnodes, int *nparts, float *tpwgts, float *ubvec, int *options, int *edgecut, idxtype *part, MPI_Comm *comm)

Description

This routine is used to compute a k-way partitioning of a *mesh* on p processors. The mesh can contain elements of different types.

Parameters

elmdist This array describes how the elements of the mesh are distributed among the processors. It is analogous to the `vtxdist` array. Its contents are identical for every processor. (See discussion in Section 4.3).

eptr, eind
These arrays specifies the elements that are stored locally at each processor. (See discussion in Section 4.3).

elmwgt This array stores the weights of the elements. (See discussion in Section 4.3).

wgtflag This is used to indicate if the graph is weighted. *wgtflag* can take one of four values:

- 0 No weights (vwgt and adjwgt are both NULL).
- 1 Weights on the edges only (vwgt is NULL).
- 2 Weights on the vertices only (adjwgt is NULL).
- 3 Weights on both the vertices and edges.

numflag This is used to indicate the numbering scheme that is used for the *elmdist*, *elements*, and *part* arrays. *numflag* can take one of two values:

- 0 C-style numbering that starts from 0.
- 1 Fortran-style numbering that starts from 1.

ncon This is used to specify the number of weights that each vertex has. It is also the number of balance constraints that must be satisfied.

ncommonnodes
This parameter determines the degree of connectivity among the vertices in the dual graph. Specifically, an edge is placed between any two elements if and only if they share at least this many nodes. This value should be greater than zero, and for most meshes a value of two will create reasonable dual graphs. However, depending on the type of elements in the mesh, values greater than two may also be valid choices. For example, for meshes containing only triangular, tetrahedral, hexahedral, or rectangular elements, this parameter can be set to two, three, four, or two, respectively.

Note that setting this parameter to a small value will increase the number of edges in the resulting dual graph and the corresponding partitioning time.

nparts This is used to specify the number of sub-domains that are desired. Note that the number of sub-domains is independent of the number of processors that call this routine.

tpwgts An array of size `ncon` x `nparts` that is used to specify the fraction of vertex weight that should be distributed to each sub-domain for each balance constraint. If all of the sub-domains are to be of the same size for every vertex weight, then each of the `ncon` x `nparts` elements should be set to a value of 1/`nparts`. If `ncon` is greater than one, the target sub-domain weights for each sub-domain are stored contiguously (similar to the `vwgt` array). Note that the sum of all of the `tpwgts` for a give vertex weight should be one.

ubvec An array of size ncon that is used to specify the imbalance tolerance for each vertex weight, with 1 being perfect balance and nparts being perfect imbalance. A value of 1.05 for each of the ncon weights is recommended.

options This is an array of integers that is used to pass parameters to the routine. Their meanings are identical to those of ParMETIS_V3_PartKway.

edgecut Upon successful completion, the number of edges that are cut by the partitioning is written to this parameter.

part This is an array of size equal to the number of locally-stored vertices. Upon successful completion the partition vector of the locally-stored vertices is written to this array. (See discussion in Section 4.4).

comm This is a pointer to the MPI communicator of the processes that call PARMETIS. For most programs this will point to MPI_COMM_WORLD.

5.2 Graph Repartitioning

ParMETIS_V3_AdaptiveRepart (idxtype *vtxdist, idxtype *xadj, idxtype *adjncy, idxtype *vwgt, idxtype *vsize, idxtype *adjwgt, int *wgtflag, int *numflag, int *ncon, int *nparts, float *tpwgts, float *ubvec, float *itr, int *options, int *edgecut, idxtype *part, MPI_Comm *comm)

Description

This routine is used to balance the work load of a graph that corresponds to an adaptively refined mesh.

Parameters

vtxdist This array describes how the vertices of the graph are distributed among the processors. (See discussion in Section 4.1). Its contents are identical for every processor.

xadj, adjncy
These store the (local) adjacency structure of the graph at each processor. (See discussion in Section 4.1).

vwgt, adjwgt
These store the weights of the vertices and edges. (See discussion in Section 4.1).

vsize This array stores the size of the vertices with respect to redistribution costs. Hence, vertices associated with mesh elements that require a lot of memory will have larger corresponding entries in this array. Otherwise, this array is similar to the `vwgt` array. (See discussion in Section 4.1).

wgtflag This is used to indicate if the graph is weighted. *wgtflag* can take one of four values:

- 0 No weights (vwgt and adjwgt are both NULL).
- 1 Weights on the edges only (vwgt is NULL).
- 2 Weights on the vertices only (adjwgt is NULL).
- 3 Weights on both the vertices and edges.

numflag This is used to indicate the numbering scheme that is used for the *vtxdist*, *xadj*, *adjncy*, and *part* arrays. *numflag* can take the following two values:

- 0 C-style numbering is assumed that starts from 0
- 1 Fortran-style numbering is assumed that starts from 1

ncon This is used to specify the number of weights that each vertex has. It is also the number of balance constraints that must be satisfied.

nparts This is used to specify the number of sub-domains that are desired. Note that the number of sub-domains is independent of the number of processors that call this routine.

tpwgts An array of size `ncon` x `nparts` that is used to specify the fraction of vertex weight that should be distributed to each sub-domain for each balance constraint. If all of the sub-domains are to be of the same size for every vertex weight, then each of the `ncon` x `nparts` elements should be set to a value of 1/`nparts`. If `ncon` is greater than one, the target sub-domain weights for each sub-domain are stored contiguously (similar to the `vwgt` array). Note that the sum of all of the `tpwgts` for a give vertex weight should be one.

ubvec An array of size `ncon` that is used to specify the imbalance tolerance for each vertex weight, with 1 being perfect balance and `nparts` being perfect imbalance. A value of 1.05 for each of the `ncon` weights is recommended.

itr This parameter describes the ratio of inter-processor communication time compared to data redistribution time. It should be set between 0.000001 and 1000000.0. If ITR is set high, a repartitioning with a low edge-cut will be computed. If it is set low, a repartitioning that requires little data redistribution will be computed. Good values for this parameter can be obtained by dividing inter-processor communication time by data redistribution time. Otherwise, a value of 1000.0 is recommended.

options This is an array of integers that is used to pass additional parameters for the routine. If *options[0]=0*, then the default values are used. If *options[0]=1*, then the remaining three elements of *options* are interpreted as follows:

- options[1] This specifies the level of information to be returned during the execution of the algorithm. Timing information can be obtained by setting this to 1. Additional options for this parameter can be obtained by looking at the the file `defs.h` in the `ParMETIS-Lib` directory. The numerical values there should be added to obtain the correct value. The default value is 0.
- options[2] This is the random number seed for the routine. The default value is 15.
- options[3] This specifies whether the sub-domains and processors are coupled or de-coupled. If the number of sub-domains desired (*i.e.*, `nparts`) and the number of processors that are being used is not the same, then these must be de-coupled. However, if `nparts` equals the number of processors, these can either be coupled or de-coupled. If sub-domains and processors are coupled, then the initial partitioning will be obtained implicitly from the graph distribution. However, if sub-domains are de-coupled from processors, then the initial partitioning needs to be obtained from the initial values assigned to the `part` array. A value of 1 indicates that sub-domains and processors are coupled and 2 indicates that these are de-coupled. The default value is 1 (coupled) if `nparts` equals the number of processors and 2 (de-coupled) otherwise.

edgecut Upon successful completion, the number of edges that are cut by the partitioning is written to this parameter.

part This is an array of size equal to the number of locally-stored vertices. Upon successful completion the partition vector of the locally-stored vertices is written to this array. (See discussion in Section 4.4). If the number of processors does not equal the number of sub-domains and/or options[3] is set to 2, then the previously computed partitioning must be passed to the routine as a parameter via this array.

comm This is a pointer to the MPI communicator of the processes that call PARMETIS. For most programs this will point to `MPI_COMM_WORLD`.

5.3 Partitioning Refinement

ParMETIS_V3_RefineKway (idxtype *vtxdist, idxtype *xadj, idxtype *adjncy, idxtype *vwgt, idxtype *adjwgt, int *wgtflag, int *numflag, int *ncon, int *nparts, float *tpwgts, float *ubvec, int *options, int *edgecut, idxtype *part, MPI_Comm *comm)

Description

This routine is used to improve the quality of an existing a k-way partitioning on p processors using the multi-level k-way refinement algorithm.

Parameters

vtxdist This array describes how the vertices of the graph are distributed among the processors. (See discussion in Section 4.1). Its contents are identical for every processor.

xadj, adjncy
These store the (local) adjacency structure of the graph at each processor. (See discussion in Section 4.1).

vwgt, adjwgt
These store the weights of the vertices and edges. (See discussion in Section 4.1).

ncon This is used to specify the number of weights that each vertex has. It is also the number of balance constraints that must be satisfied.

nparts This is used to specify the number of sub-domains that are desired. Note that the number of sub-domains is independent of the number of processors that call this routine.

wgtflag This is used to indicate if the graph is weighted. *wgtflag* can take one of four values:

0 No weights (vwgt and adjwgt are both NULL).
1 Weights on the edges only (vwgt is NULL).
2 Weights on the vertices only (adjwgt is NULL).
3 Weights on both the vertices and edges.

numflag This is used to indicate the numbering scheme that is used for the *vtxdist*, *xadj*, *adjncy*, and *part* arrays. *numflag* can take the following two values:

0 C-style numbering is assumed that starts from 0
1 Fortran-style numbering is assumed that starts from 1

tpwgts An array of size `ncon` x `nparts` that is used to specify the fraction of vertex weight that should be distributed to each sub-domain for each balance constraint. If all of the sub-domains are to be of the same size for every vertex weight, then each of the `ncon` x `nparts` elements should be set to a value of 1/`nparts`. If `ncon` is greater than one, the target sub-domain weights for each sub-domain are stored contiguously (similar to the `vwgt` array). Note that the sum of all of the `tpwgts` for a give vertex weight should be one.

ubvec An array of size `ncon` that is used to specify the imbalance tolerance for each vertex weight, with 1 being perfect balance and `nparts` being perfect imbalance. A value of 1.05 for each of the `ncon` weights is recommended.

options This is an array of integers that is used to pass parameters to the routine. Their meanings are identical to those of ParMETIS_V3_PartKway.

edgecut Upon successful completion, the number of edges that are cut by the partitioning is written to this parameter.

part This is an array of size equal to the number of locally-stored vertices. Upon successful completion the partition vector of the locally-stored vertices is written to this array. (See discussion in Section 4.4).

comm This is a pointer to the MPI communicator of the processes that call PARMETIS. For most programs this will point to `MPI_COMM_WORLD`.

5.4 Fill-reducing Orderings

ParMETIS_V3_NodeND (idxtype *vtxdist, idxtype *xadj, idxtype *adjncy, int *numflag, int *options, idxtype *order, idxtype *sizes, MPI_Comm *comm)

Description

This routine is used to compute a fill-reducing ordering of a sparse matrix using multilevel nested dissection.

Parameters

vtxdist This array describes how the vertices of the graph are distributed among the processors. (See discussion in Section 4.1). Its contents are identical for every processor.

xadj, adjncy
These store the (local) adjacency structure of the graph at each processor (See discussion in Section 4.1).

numflag This is used to indicate the numbering scheme that is used for the *vtxdist*, *xadj*, *adjncy*, and *order* arrays. *numflag* can take the following two values:

0 C-style numbering is assumed that starts from 0

1 Fortran-style numbering is assumed that starts from 1

options This is an array of integers that is used to pass parameters to the routine. Their meanings are identical to those of ParMETIS_V3_PartKway.

order This array returns the result of the ordering (described in Section 4.4).

sizes This array returns the number of nodes for each sub-domain and each separator (described in Section 4.4).

comm This is a pointer to the MPI communicator of the processes that call PARMETIS. For most programs this will point to `MPI_COMM_WORLD`.

Note

ParMETIS_V3_NodeND requires that the number of processors be a power of 2.

5.5 Mesh to Graph Translation

ParMETIS_V3_Mesh2Dual (idxtype *elmdist, idxtype *eptr, idxtype *eind, int *numflag,
int *ncommonnodes, idxtype **xadj, idxtype **adjncy, MPI_Comm *comm)

Description

This routine is used to construct a distributed graph given a distributed mesh. It can be used in conjunction with other routines in the PARMETIS library. The mesh can contain elements of different types.

Parameters

elmdist This array describes how the elements of the mesh are distributed among the processors. It is analogous to the `vtxdist` array. Its contents are identical for every processor. (See discussion in Section 4.3).

eptr, eind
These arrays specifies the elements that are stored locally at each processor. (See discussion in Section 4.3).

numflag This is used to indicate the numbering scheme that is used for the *elmdist*, *elements*, *xadj*, *adjncy*, and *part* arrays. *numflag* can take one of two values:

0 C-style numbering that starts from 0.

1 Fortran-style numbering that starts from 1.

ncommonnodes
This parameter determines the degree of connectivity among the vertices in the dual graph. Specifically, an edge is placed between any two elements if and only if they share at least this many nodes. This value should be greater than zero, and for most meshes a value of two will create reasonable dual graphs. However, depending on the type of elements in the mesh, values greater than two may also be valid choices. For example, for meshes containing only triangular, tetrahedral, hexahedral, or rectangular elements, this parameter can be set to two, three, four, or two, respectively.

Note that setting this parameter to a small value will increase the number of edges in the resulting dual graph and the corresponding partitioning time.

xadj, adjncy
Upon the successful completion of the routine, pointers to the constructed `xadj` and `adjncy` arrays will be written to these parameters. (See discussion in Section 4.1).

comm This is a pointer to the MPI communicator of the processes that call PARMETIS. For most programs this will point to `MPI_COMM_WORLD`.

Note

This routine can be used in conjunction with ParMETIS_V3_PartKway, ParMETIS_V3_PartGeomKway, or ParMETIS_V3_AdaptiveRepart. It typically runs in half the time required by ParMETIS_V3_PartKway.

6 Hardware & Software Requirements, and Contact Information

PARMETIS is written in ANSI C and uses MPI for inter-processor communication. Instructions on how to build PARMETIS are available in the `INSTALL` file. In the directory called `Graphs`, you will find programs that tests if PARMETIS was built correctly. Also, a header file called `parmetis.h` is provided that contains prototypes for the functions in PARMETIS.

In order to use PARMETIS in your application you need to have a copy of the serial METIS library and link your program with both libraries (*i.e.*, `libparmetis.a` and `libmetis.a`). Note that the PARMETIS package already contains the source code for the METIS library. The included Makefiles automatically construct both libraries.

PARMETIS have been extensively tested on a number of different parallel computers. However, even though PARMETIS contains no known bugs, this does not mean that all of its bugs have been found and fixed. If you have any problems, please send email to *metis@cs.umn.edu* with a brief description of the problem.

References

[1] R. Biswas and R. Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Applied Numerical Mathematics*, 13:437–452, 1994.

[2] C. Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *In Proc. 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[3] J. Fingberg, A. Basermann, G. Lonsdale, J. Clinckemaillie, J. Gratien, and R. Ducloux. Dynamic load-balancing for parallel structural mechanics simulations with DRAMA. *ECT2000*, 2000.

[4] G. Karypis and V. Kumar. A coarse-grain parallel multilevel k-way partitioning algorithm. In *Proceedings of the 8th SIAM conference on Parallel Processing for Scientific Computing*, 1997.

[5] G. Karypis and V. Kumar. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, version 4.0. Technical report, Univ. of MN, Dept. of Computer Sci. and Engr., 1998.

[6] G. Karypis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *Proc. Supercomputing '98*, 1998.

[7] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1), 1998.

[8] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning scheme for irregular graphs. *Siam Review*, 41(2):278–300, 1999.

[9] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[10] L. Oliker and R. Biswas. PLUM: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52(2):150–177, 1998.

[11] A. Patra and D. Kim. Efficient mesh partitioning for adaptive *hp* finite element meshes. Technical report, Dept. of Mech. Engr., SUNY at Buffalo, 1999.

[12] A. Pothen, H. Simon, L. Wang, and S. Bernard. Towards a fast implementation of spectral nested dissection. In *Supercomputing '92 Proceedings*, pages 42–51, 1992.

[13] K. Schloegel, G. Karypis, and V. Kumar. A new algorithm for multi-objective graph partitioning. In *Proc. EuroPar '99*, pages 322–331, 1999.

[14] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. In *Proc. EuroPar-2000*, 2000. Accepted as a Distinguished Paper.

[15] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proc. Supercomputing 2000*, 2000.

[16] K. Schloegel, G. Karypis, and V. Kumar. Wavefront diffusion and LMSR: Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Transactions on Parallel and Distributed Systems*, 12(5):451–466, 2001.

[17] J. Watts, M. Rieffel, and S. Taylor. A load balancing technique for multi-phase computations. *Proc. of High Performance Computing '97*, pages 15–20, 1997.