# METIS*

## Unstructured Graph Partitioning and Sparse Matrix Ordering System

### Version 2.0

GEORGE KARYPIS AND VIPIN KUMAR

University of Minnesota, Department of Computer Science
Minneapolis, MN 55455

{karypis, kumar}@cs.umn.edu

August 26, 1995

Metis [MEE tis]: *'Metis' is the Greek word for wisdom. Metis was a titaness in Greek mythology. She was the consort of Zeus and the mother of Athena. She presided over all wisdom and knowledge.*

# 1 Introduction

Graph partitioning has extensive applications in many areas, including scientific computing, VLSI design, and task scheduling. The problem is to partition the vertices of a graph in $p$ roughly equal parts, such that the number of edges connecting vertices in different parts is minimized. For example, the solution of a sparse system of linear equations $Ax = b$ via iterative methods on a parallel computer gives rise to a graph partitioning problem. A key step in each iteration of these methods is the multiplication of a sparse matrix and a (dense) vector. Partitioning the graph corresponding to the matrix $A$, significantly reduces the amount of communication required by this step [25]. If parallel direct methods are used to solve a sparse system of equations, then a graph partitioning algorithm can be used to compute a fill reducing ordering that lead to high degree of concurrency in the factorization phase [25, 10]. The multiple minimum degree ordering used almost exclusively in serial direct methods is not suitable for parallel direct methods, as it provides very little concurrency in the parallel factorization phase.

METIS is a set of programs that implement the various algorithms described in [22, 23]. The advantages of METIS compared to other similar packages are the following:

☞ **Provides high quality partitions!**
The partitions produced by METIS are consistently 10% to 50% better than those produced by spectral partitioning algorithms [1, 19], and 5% to 15% better than those produced by Chaco multilevel [20, 19].

☞ **It is extremely fast!**
In our experiments, on a wide range of graphs, we found that the multilevel recursive bisection algorithms

*METIS is copyrighted by the regents of the University of Minnesota. This work was supported by IST/BMDO through Army Research Office contract DA/DAAH04-93-G-0080, and by Army High Performance Computing Research Center under the auspices of the Department of the Army, Army Research Laboratory cooperative agreement number DAAH04-95-2-0003/contract number DAAH04-95-C-0008, the content of which does not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred. Access to computing facilities was provided by Minnesota Supercomputer Institute, Cray Research Inc, and by the Pittsburgh Supercomputing Center. Related papers are available via WWW at URL: **http://www.cs.umn.edu/~karypis**

implemented by METIS are 10 to 40 times faster than multilevel spectral bisection, and 2 to 6 times faster than Chaco multilevel. Furthermore, the multilevel $k$-way partitioning algorithms implemented by METIS are 40 to 160 times faster than multilevel spectral bisection and 8 to 16 times faster than Chaco multilevel. Graphs with over half a million vertices can be partitioned in 256 parts, in under a minute on scientific workstations. The run time of METIS is comparable to (or even smaller than) the run time of geometric partitioning algorithms that often produce much worse partitions.

☞ **Provides low fill orderings!**
The orderings produced by METIS are significantly better than those produced by multiple minimum degree, particularly for large finite element graphs. Furthermore, unlike multiple minimum degree, the elimination trees produced by METIS are suited for parallel direct factorization.

The rest of this paper is organized as follows: Section 2 briefly describes the various ideas and algorithms implemented in METIS. Section 3 describes the user interface to the METIS graph partitioning and sparse matrix ordering packages. Sections 4 and 5 describe the formats of the input and output files used by METIS. Section 6 describes the stand-alone library that implements the various algorithms implemented in METIS. Section 7 describes the system requirements for the METIS package. Appendix A describes and compares various graph partitioning algorithms that are extensively used.

# 2 Multilevel Graph Partition

The basic idea behind the multilevel graph partition algorithms implemented in METIS is very simple. The graph $G$ is first coarsened down to a few hundred vertices, a bisection of this much smaller graph is computed, and then this partition is projected back towards the original graph (finer graph), by periodically refining the partition. Since the finer graph has more degrees of freedom, such refinements usually decrease the edge-cut. This process, is graphically illustrated in Figure 1.

In the rest of this section we briefly describe the various phases of the multilevel algorithm. The reader should refer to [22] for further details.

## 2.1 Coarsening Phase

During the coarsening phase, a sequence of smaller graphs $G_l = (V_l, E_l)$, is constructed from the original graph $G_0 = (V_0, E_0)$ such that $|V_l| > |V_{l+1}|$. Graph $G_{l+1}$ is constructed from $G_l$ by finding a maximal matching $M_l \subseteq E_l$ of $G_l$ and collapsing together the vertices that are incident on each edge of the matching. In this process no more than two vertices are collapsed together because a matching of a graph is a set of edges, no two of which are incident on the same vertex. Vertices that are not incident on any edge of the matching are simply copied over to $G_{l+1}$.

When vertices $v, u \in V_l$ are collapsed to form vertex $w \in V_{l+1}$, the weight of vertex $w$ is set equal to the sum of the weights of vertices $v$ and $u$, while the edges incident on $w$ is set equal to the union of the edges incident on $v$ and $u$ minus the edge $(v, u)$. If there is an edge that is incident to both on $v$ and $u$, then the weight of this edge is set equal to the sum of the weights of these edges. Thus, during successive coarsening levels, the weight of both vertices and edges increases.

Maximal matchings can be computed in different ways [22]. The method used to compute the matching greatly affects both the quality of the bisection, and the time required during the uncoarsening phase. METIS implements four different matching schemes. Here we briefly describe two of the matching schemes implemented in METIS. Information about the other two schemes can be found in [22].

The first scheme, which is called ***random matching*** (RM), computes the maximal matching by using a randomized algorithm [3, 20]. The RM scheme works as follows. The vertices of the graph are visited in random order. If a vertex $u$ has not been matched yet, then an unmatched adjacent vertex $v$ is randomly selected and the edge $(u, v)$ is included in the matching. If there is no unmatched adjacent vertex $v$, then vertex $u$ remains unmatched. The second scheme, which we call ***heavy-edge matching*** (HEM), computes a matching $M_l$, such that the weight of the edges in $M_l$ is high. The heavy-edge matching is computed using a randomized algorithm similar to the one used for RM. The vertices are again visited in random order. However, instead of randomly matching a vertex with one of its adjacent unmatched vertices, HEM matches it with the unmatched vertex that is connected with the heavier edge. As a result, the HEM scheme reduces the sum of the weights of the edges in the coarser graph by a larger amount than RM. In [22],

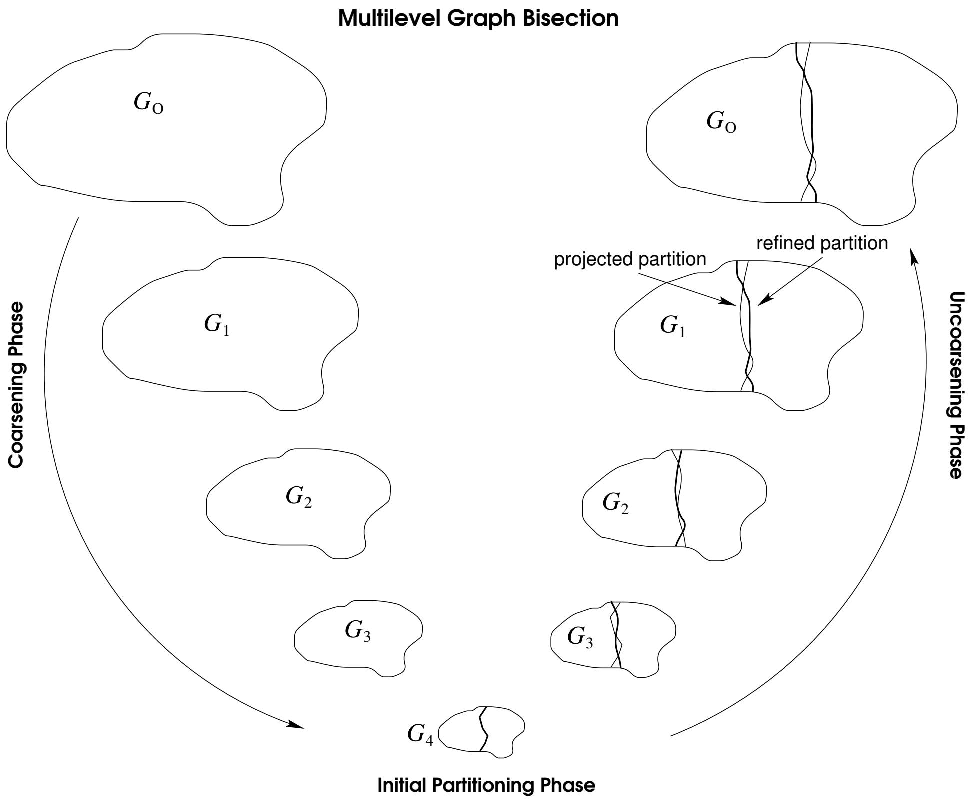



**Figure 1**: The various phases of the multilevel graph bisection. During the coarsening phase, the size of the graph is successively decreased; during the initial partitioning phase, a bisection of the smaller graph is computed; and during the uncoarsening phase, the bisection is successively refined as it is projected to the larger graphs. During the uncoarsening phase the light lines indicate projected partitions, and dark lines indicate partitions that were produced after refinement.

we experimentally evaluated both the RM and HEM matching schemes. We found that the HEM scheme produces consistently better results than RM, and the amount of time spent in refinement is less than that of RM.

## 2.2 Partitioning Phase

The second phase of a multilevel algorithm is to compute a minimum edge-cut bisection of the coarse graph $G_k = (V_k, E_k)$ such that each part contains roughly half of the vertex weight of the original graph. Since during coarsening, the weights of the vertices and edges of the coarser graph were set to reflect the weights of the vertices and edges of the finer graph, $G_k$ contains sufficient information to intelligently enforce the balanced partition and the minimum edge-cut requirements.

A partition of $G_k$ can be obtained using various algorithms such as (a) spectral bisection [32, 31, 1, 18], (b) geometric bisection [28, 27] (if coordinates are available), and (c) combinatorial methods [24, 2, 9, 10, 12, 4]. Since the size of the coarser graph $G_k$ is small (*i.e.*, $|V_k| < 100$), this step takes a small amount. METIS implements four different schemes for partitioning the coarsest graph, that are evaluated in [22]. Three of these algorithms are based on graph growing heuristics, and the other one is based on spectral bisection. We found that all four algorithms produce fairly similar partitions, with the graph growing heuristics doing usually better.

## 2.3 Uncoarsening Phase

During the uncoarsening phase, the partition of the coarsest graph $G_k$ is projected back to the original graph by going through the graphs $G_{k-1}, G_{k-2}, \ldots, G_1$. Since each vertex $u \in V_l$ contains a distinct subset $U$ of vertices of $V_{l-1}$, projecting the partition of $G_l$ to $G_{l-1}$ is done by simply assigning the vertices in $U$ to the same part that vertex $u$ belongs to.

Furthermore, even if the partition of $G_l$ is at a local minima, the projected partition of $G_{l-1}$ may not be at a local minima. Since $G_{l-1}$ is finer, it has more degrees of freedom that can be used to further improve the partition and thus decrease the edge-cut. Hence, it may still be possible to improve the projected partition of $G_{l-1}$ by local refinement heuristics. For this reason, after projecting a partition, a partition refinement algorithm is used. The basic purpose of a partition refinement algorithm is to select two subsets of vertices, one from each part such that when swapped the resulting partition has smaller edge-cut. Specifically, if $A$ and $B$ are the two parts of the bisection, a refinement algorithm selects $A' \subset A$ and $B' \subset B$ such that $A \backslash A' \cup B'$ and $B \backslash B' \cup A'$ is a bisection with a smaller edge-cut.

A class of algorithms that tend to produce very good results are those that are based on the Kernighan-Lin (KL) partition algorithm [24, 7, 20]. The KL algorithm is iterative in nature. It starts with an initial partition and in each iteration it finds subsets $A'$ and $B'$ with the above properties. If such subsets exist, then it moves them to the other part and this becomes the partition for the next iteration. The algorithm continues by repeating the entire process. If it cannot find two such subsets, then the algorithm terminates, since the partition is at a local minima and no further improvement can be made by the KL algorithm. The KL algorithm has been found to be effective in finding locally optimal partitions when it starts with a fairly good initial partition. Since the projected partition is already a good partition, KL substantially decreases the edge-cut within a small number of iterations [22].

The KL algorithm implemented in METIS is similar to that described in [7]. The KL algorithm computes for each vertex $v$ a quantity called ***gain*** which is the decrease (or increase) in the edge-cut if $v$ is moved to the other part. The algorithm then proceeds by repeatedly selecting a vertex $v$ with the largest gain from the larger part and moves it to the other part. After moving $v$, (a) $v$ is marked so it will not be considered again in the same iteration, and (b) the gains of the vertices adjacent to $v$ are updated to reflect the change in the partition. The algorithm terminates when the edge-cut does not decrease after $x$ number of vertex moves. Since, the last $x$ vertex moves do not decrease the edge-cut but they may increase it, the last $x$ number of moves are undone.

The KL algorithm, as described, may require a relatively large number of iterations before it converges. In [22] we used a variant that performs only a single iteration. This scheme which is called greedy refinement (GR), was found to be significantly faster than KL and produced partitions that were only slightly worse than KL. Furthermore, in [22] we implemented another variation of the algorithm, that only allows vertices along the boundary of the bisection to be swapped. Even though, this might seem very restrictive, our experiments in [22] show that boundary refinement performs at least as well as KL, requiring significantly less time. This is because most of the nodes swapped by the KL algorithm are along the boundary of the cut, which is defined to be the vertices that have edges that are cut by the partition. We will refer to this algorithm as ***boundary Kernighan-Lin*** (BKLR).

# 3 Usage of the METIS Package

METIS provides two different interfaces. The first interface, called ***simple interface***, is for those users that want to use METIS to either partition or order a sparse matrix without getting into the details of the underlying algorithms implemented by METIS. The second interface, called ***advanced interface***, is for the users that want to experiment with the different parameters of the algorithm.

The simple interface is provided via the programs `pmetis`, `kmetis`, and `ometis`. The programs `pmetis` and `kmetis` implement $k$-way partitioning, while the program `ometis` performs ordering. `pmetis` and `kmetis` are invoked by providing two arguments at the command line. The first argument is the name of the file that stores the graph, while the second argument is the number of partitions that is desired (any number of partitions is allowed). The graph file must be stored in the format described in Section 4. Upon successful execution, `pmetis` and `kmetis` display statistics regarding the number of edges being cut, and the amount of time taken to perform the partition, and it stores the computed partition in a file, as described in Section 5. Figure 2 shows the output of the `pmetis`, and Figure 3 shows the output of the `kmetis` for a sample graph. From these figures we see that `pmetis` and `kmetis` initially print information about the graph, such as its name and its size (the number of vertices and edges), and then the default parameters used by the multilevel algorithm. Next, they print the edge-cut of the $k$-way partition. If $k$ is a power of 2, `pmetis` also prints the edge-cuts for the intermediate partitions. In addition to the edge-cut, both `pmetis` and `kmetis` print the partition balance. For a $k$-way partition of a graph with $n$ vertices, let $m$ be the size of the largest part produced by the $k$-way partitioning algorithm. The partition balance is defined as $km/n$, and is essentially the load imbalance induced by non-equal partitions. `pmetis` produces partitions that are perfectly balanced at each bisection level, however, some small load imbalance may result due to the $\log k$ levels of recursive bisection. In general, the load imbalance is less than 1%. `kmetis` produces partitions that are not perfectly balanced, and the algorithms limits the load imbalance to 3%.

Finally, both `pmetis` and `kmetis` show the time that was taken by the various phases of the algorithm. All times are in seconds. In this particular example, the 256-way partition, took `pmetis` a total of 31.920 seconds, of which 25.680 seconds was taken by the partitioning algorithm itself, whereas took `kmetis` a total of 15.360 seconds, of which 9.170 was taken by the partitioning algorithm itself.

`ometis` is invoked by providing only a single argument, that of the name of the file that stores the graph to be reordered. Upon successful execution, `ometis` displays statistics regarding the fill produced by the reordering, and the amount of time taken to perform the partition. Also, it stores the computed permutation in a file as described in Section 5. Figure 4 shows the output of the `ometis` for a sample graph. After finding the fill reducing ordering, `ometis` symbolically factors the matrix to determine both the number of nonzeros required to store the symbolically factored matrix, and the number of operations (OPC) required to directly factor the matrix using Cholesky factorization.

**Note** Both the advanced and simple interfaces are implemented by the simple executable. The program selects which interface to choose by the name of the executable used to invoke METIS. Thus, the name of the executables should not be modified by the user.

### 3.1 Advanced Interface

The advanced interface of METIS is provided to allow the experienced user to experiment with different algorithms implemented for the various phases of the multilevel graph partitioning algorithm. Detailed information about the meaning of these various algorithms and their effect on the quality of the produced partitions can be found in [22].

The advanced interface of METIS is provided by the program `metis`, and requires 8 command line arguments. The syntax of METIS is as follows:

```
metis GraphFile Nparts CoarsenTo Mtype Rtype IPtype OPtype Options
```

The meaning of the various parameters is as follows:

**GraphFile** This is the name of the file that stores the graph *G*, in the format described in Section 4.

**Nparts** The number of parts that the graph needs to be partitioned. Nparts must be greater than one, and its value is ignored when METIS is used to order *G*.

**CoarsenTo** This is the number of vertices that *G* must be coarsened down, during the coarsening phase. The default value used by `pmetis` and `ometis` is 100 vertices, while a value of 2000 is used by `kmetis`.

**Mtype** This is the type of matching to be used during the coarsening phase. *Mtype* is an integer, whose values and their meaning are given in the following table:

| Mtype Value | Abbrev. | Matching Scheme | `pmetis` | `kmetis` | `ometis` |
|---|---|---|---|---|---|
| 1 | RM | Random | ✓✓ | ✓✓ | ✓✓ |
| 2 | HEM | Heavy-edge | ✓✓ | ✓✓ | ✓✓ |
| 3 | LEM | Light-edge | ✓✓ | ✓ | ✓✓ |
| 4 | HCM | Heavy-clique | ✓✓ | ✓✓ | ✓✓ |
| 5 | HEM* | Modified Heavy-edge | ✓ | ✓✓ | ✓✓ |
| 11 | SRM | Sorted Random | ✓✓ | ✓✓ | ✓✓ |
| 21 | SHEM | Sorted Heavy-edge | ✓✓ | ✓✓ | ✓✓ |
| 51 | SHEM* | Sorted Modified Heavy-edge | ✓ | ✓✓ | ✓✓ |

Out of these schemes the first five of them are the main matching schemes that are described in detail in [22, 23]. The last three matching schemes are variations of RM, HEM and HEM*. These schemes before proceeding to find a matching using either RM, HEM, and HEM*, they first sort the vertices in non-decreasing vertex degree, and they use this order to find the matchings. Our experiments have shown that the sorting variants tend to find larger matchings than their non-sorting counterparts. When one of the sorting matching schemes is used, METIS performs the first few levels of coarsening using the non-sorting variants and it the switches to the sorting variants. It does that to save time, since sorting is relatively expensive. Along with each matching scheme, the above table also shows whether or not a particular matching scheme is applicable to one of the three operations implemented in METIS. Both '✓✓' and '✓'

```
prompt% pmetis brack2.graph 256

*******************************************************************
 METIS 2.0   Copyright 1995, Regents of the University of Minnesota

Graph Information --------------------------------------------------
   Graph: brack2.graph, Size: 62631,  733118, Parts: 256, Cto: 100
 Options: SHEM, BGKLR, GGPKL, Rec-Partition

Recursive Partitioning... ------------------------------------------
  Edge-Cuts:
    2-way   4-way   8-way  16-way  32-way  64-way 128-way 256-way
      757    3563    8764   14292   21047   29927   43288   60653
  Balance: 1.010

Timing Information -------------------------------------------------
 Multilevel:              25.680
     Coarsening:                  13.680
     Initial Partition:            2.880
     Uncoarsening:                 6.510
     Splitting:                    2.570
 I/O:                      4.450
 Total:                   31.920
*******************************************************************
```

**Figure 2**: Output of `pmetis` for graph *brack2.graph* and a 256-way partition.

```
prompt% kmetis brack2.graph 256

*******************************************************************
 METIS 2.0   Copyright 1995, Regents of the University of Minnesota

Graph Information --------------------------------------------------
   Graph: brack2.graph, Size: 62631,  733118, Parts: 256, Cto: 2000
 Options: SHEM, BGR, GGPKL, K-Partition

K-way Partitioning... ----------------------------------------------
  256-way Edge-Cut:   59883, Balance: 1.026

Timing Information -------------------------------------------------
 Multilevel:               9.170
     Coarsening:                   2.820
     Initial Partition:            4.050
     Uncoarsening:                 2.290
 I/O:                      4.430
 Total:                   15.360
*******************************************************************
```

**Figure 3**: Output of `kmetis` for graph *brack2.graph* and a 256-way partition.

```
prompt% ometis brack2.graph

*******************************************************************
 METIS 2.0   Copyright 1995, Regents of the University of Minnesota

Graph Information ---------------------------------------------------
   Graph: brack2.graph, Size: 62631,  733118, Parts:   0, Cto: 100
 Options: SHEM, BGKLR, GGGP, Order

Ordering... ---------------------------------------------------------
  Symbolic factorization...
  Nonzeros: 7216188      OPC: 2.5465e+09

Timing Information --------------------------------------------------
 Multilevel:                24.040
     Coarsening:                    12.870
     Initial Partition:              1.010
     Uncoarsening:                   5.970
     Splitting:                      2.900
 I/O:                        4.420
 Total:                     34.080
*******************************************************************
```

**Figure 4**: Output of `ometis` for graph *brack2.graph*.

indicate that this matching scheme is supported, however, '✓' indicates that this particular matching scheme either requires a lot of time, or does not produce very good results.

Out of these matching schemes, HEM, HCM, and HEM* perform consistently better than either RM or LEM. In particular, the analysis presented in [21] shows that HEM does significantly better than RM, especially for 3D finite element graphs. On the other hand, the LEM matching scheme significantly increases the average degree of the coarser graphs, a feature that helps certain refinement algorithms to perform quite well. However, when LEM is used during coarsening, both the time spent during coarsening and the time spent during uncoarsening is fairly high. SHEM is the matching scheme that is used by `pmetis`, `kmetis`, and `ometis`.

**Rtype** This is the type of refinement policy to use during the uncoarsening phase. *Rtype* is an integer whose values and their meaning is as follows:

| Rtype Value | Abbrev. | Refinement Policy | `pmetis` | `kmetis` | `ometis` |
|---|---|---|---|---|---|
| 1 | GR | Greedy | ✓ | ✗ | ✓ |
| 2 | KLR | Kernighan-Lin | ✓ | ✗ | ✓ |
| 3 | GKLR | Combination of GR and KLR | ✓ | ✗ | ✓ |
| 11 | BGR | Boundary Greedy | ✓✓ | ✓✓ | ✓✓ |
| 12 | BKLR | Boundary Kernighan-Lin | ✓✓ | ✓✓ | ✓✓ |
| 13 | BGKLR | Combination of BGR and BKLR | ✓✓ | ✗ | ✓✓ |
| 20 | NR | No refinement | ✓ | ✓ | ✓ |

Along with each refinement scheme, the above table also shows whether or not a particular refinement scheme is applicable to one of the three operations implemented in METIS. Both '✓✓' and '✓' indicate that this refinement scheme is supported; however, '✓' indicates that this particular refinement scheme either requires a lot of time, or does not produce very good results. The '✗' indicates that this refinement scheme is not supported.

The boundary refinement algorithms perform better than their non-boundary counterparts. Furthermore, the boundary refinement algorithms require significantly less time. BGR is the fastest refinement algorithm, but tends to perform worse than either BKLR or BGKLR. Both BKLR and BGKLR, provide very good refinement and they roughly require the same amount of time. BGKLR is the refinement scheme used by `pmetis` and `ometis`. Experiments show that BGKLR performs better than BKLR when the graph is partitioned into many parts (*i.e.*, more than 64 parts). On the other hand, BKLR tends to perform slightly better than BGKLR for small number of partitions. However, the difference in quality is very small (less than 2%). You may want to experiment between BKLR and BGKLR to see which one performs better for your graphs.

BGR is the refinement scheme used by `kmetis`. Experiments show that for most graphs, particularly those that correspond to finite element meshes, BGR performs significantly better than BKLR, and it also require much less time.

**IPtype** This is the type of bisection algorithm to use at the coarsest graph. *IPtype* is an integer whose values and their meaning is as follows:

| IPtype Value | Abbrev. | Bisection algorithm |
|---|---|---|
| 1 | GGP | Graph growing partition |
| 2 | GGGP | Greedy graph growing partition |
| 3 | EIG | Spectral bisection |
| 4 | GGPKL | Graph growing followed by boundary Kernighan-Lin |

GGP is the fastest partitioning algorithm and also tends to perform worse than GGGP and GGPKL. EIG tends to perform worse than GGGP and GGPKL and requires more time, particularly when the coarsest graph is quite large. GGGP and GGPKL require roughly the same amount of time, and they tend to perform quite similarly. You may want to experiment between GGGP and GGPKL to see which one performs better for your graphs. GGPKL is the partitioning scheme used by `pmetis` and GGGP is the partitioning scheme used by `ometis`. This parameter is not used by `kmetis`.

**OPtype** This is the type of operation to be performed by METIS. *OPtype* is an integer whose values and their meaning is as follows:

| OPtype Value | Operation performed |
|---|---|
| 1 | Graph partitioning using recursive bisection |
| 2 | Graph partitioning using $k$-way partitioning |
| 3 | Matrix ordering |

**Options** This is used to request METIS to perform certain additional operations, such as write partition and permutation files, print timing information, and print debugging information. The value of *Options* is computed as the sum of codes associated with each option of METIS. The various options and their values are:

| Code | Option requested | `pmetis` | `kmetis` | `ometis` |
|---|---|---|---|---|
| 1 | Write the resulting partition or permutation vectors to a file | ✓✓ | ✓✓ | ✓✓ |
| 2 | Print timing statistics for various phases of the algorithm (P,O) | ✓✓ | ✓✓ | ✓✓ |
| 4 | Show information during the coarsening phase | ✓✓ | ✓ | ✓✓ |
| 8 | Show information during the initial partition phase | ✓✓ | ✓ | ✓✓ |
| 16 | Show information during the refinement phase | ✓✓ | ✓ | ✓✓ |
| 32 | Show the size of the partitions at the end | ✓✓ | ✓✓ | ✗ |
| 64 | Show the size of the vertex separators at each level | ✗ | ✗ | ✓✓ |
| 128 | Show the balance of a subtree-to-subcube mapping of the elimination tree | ✗ | ✗ | ✓✓ |
| 256 | Show information during the $k$-way refinement phase | ✗ | ✓✓ | ✗ |

For example, if we want both the partition to be written on a file and also to display timing information, the value of *Options* should be *3*. A value of *195*, in addition to the above also shows the size of the separators and the balance of the ordering.

Options marked with '✓✓', '✓', and '✗' for each one of the three operations performed by METIS. The '✓' means that the information printed out will not be very meaningful.

**Note** Some of the options may generate a lot of output. Use them with caution.

Figure 5 shows some sample output of the advanced interface of METIS. In the first example, `metis` was used to obtain an ordering of the graph *brack2.graph*, using HEM during coarsening, BKLR during refinement, GGGP for bisecting the coarsest graph, and requesting that both the times and the degree of balance of the resulting elimination tree to be displayed. The degree of balance is computed as an upper bound on the efficiency of a parallel algorithm that uses subtree-to-subcube mapping [13] (*i.e.*, assuming communication cost to be zero). For instance, a balance factor of 0.654 on 64 processors, indicates that due to load imbalance alone, parallel Cholesky factorization algorithm on 64 processors will have an upper bound on the efficiency of 0.654. In the second example of Figure 5, METIS was used to obtain a 128-way partition of *brack2.graph*, using HCM during coarsening, BGR during refinement, GGP for bisecting the coarsest graph, and requesting only timing information.

# 4 Format of Graph Input File

The primary input of METIS is the graph to be partitioned or ordered. This graph is stored in a file and is supplied to METIS as one of the command line parameters. The format of this graph file is similar to the format used by the Chaco graph partitioning package [19].

A graph $G = (V, E)$ with $n$ vertices and $m$ edges is stored in a plain text file that contains $n + 1$ lines. The first line contains information about the size and the type of the graph, while the remaining $n$ lines contain information for each vertex of $G$. Any line that starts with '%' is a comment line and is skipped.

The first line contains either two or three integers. The first integer is the number of vertices ($n$), the second is the number of edges ($m$), and the third integer (*fmt*) contains information about the type of the graph. In particular, depending on the value of *fmt*, the graph $G$ can have weights on the edges, vertices, or both. In the case that $G = (V, E)$ is unweighted (*i.e.*, all vertices and edges have the same weight), *fmt* is omitted.

After this first line, the remaining $n$ lines store the adjacency lists for each vertex—one line per vertex. In particular, the $i$th line (excluding comment lines) contains the adjacency list of the $i - 1$st vertex. Note that the numbering starts from 1 (not from 0 as it often done in C).

The simplest format for a graph $G$ is when the weight of all vertices and the weight of all the edges is the same. This format is illustrated in Figure 6(a). Note, the optional *fmt* parameter is not required in this case.

However, there are cases in which the edges in $G$ have different weights. This is accommodated as shown in Figure 6(b). Now, the adjacency list of each vertex contains in addition to the vertices that is connected with, the weight of the edges. If $v$ has $k$ vertices adjacent to it, then the graph file contains $2 * k$ numbers, each pair of numbers stores the vertex that $v$ is connected to, and the weight of the edge. Note, edge weights are integer quantities. Furthermore, note that the *fmt* parameter is equal to 1, indicating the fact that $G$ has weights on the edges.

Finally, in addition to having weights on the edges, weights on the vertices are also allowed, as illustrated in Figure 6(c). In this case, the value of *fmt* is equal to 11, and each line of the graph file first stores the weight of the vertex, and then the weighted adjacency list. As was the case with edge weights, vertex weights are integer quantities.

Note that the format of the graph file accepted by METIS is a subset of that accepted by Chaco. METIS does not support graph files in which the adjacency lists in successive lines do not correspond to adjacency lists of successive vertices.

# 5 Format of Output Files

The output of METIS is either a partition or a permutation file, depending on whether METIS is used for graph partitioning or for sparse matrix ordering.

The partition file of a graph with $n$ vertices, consists of $n$ lines with a single number per line. The $i$th line of the file contains the partition number that the $i$th vertex belongs to. Partition numbers start from 0, and partitions are assigned in a hypercube fashion, as illustrated in Figure 7. If `foo.graph` is the name of the file storing the graph, the partition for a $k$-way partition is stored in a file named `foo.graph.k.part`.

The permutation file of a graph with $n$ vertices also consists of $n$ lines with a single number per line. The $i$th line of the permutation file contains the new order of the $i$th vertex of the graph. Let $A$ be the matrix whose nonzero elements correspond to graph $G$, and let $P$ be the permutation vector produced by METIS. Row (column) $i$ of $A$ is mapped to row (column) $P[i]$ of the reordered matrix $A'$. If `foo.graph` is the name of the file storing the graph, the fill reducing permutation is stored in a file named `foo.graph.perm`.

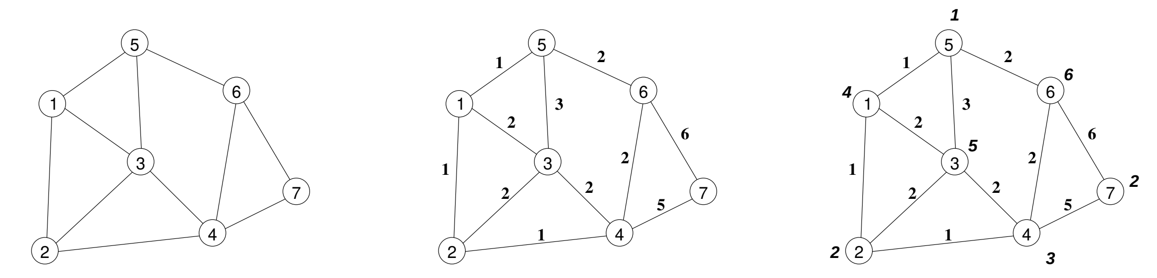
# 6 METIS Stand-Alone Library Interface

The various algorithms implemented by METIS can be also directly accessed from a C or Fortran program, using the stand-alone METISlib library. METISlib provides three routines that the user can call to invoke the three major algorithms

```
prompt% metis brack2.graph 2 100 2 12 2 3 130

*******************************************************************
 METIS 2.0   Copyright 1995, Regents of the University of Minnesota

Graph Information ---------------------------------------------------
   Graph: brack2.graph, Size: 62631,  733118, Parts:   2, Cto: 100
 Options: HEM, BKLR, GGGP, Order

Ordering... ---------------------------------------------------------
  Symbolic factorization...
  Nonzeros: 7295136      OPC: 2.5694e+09
  Balance:   2-PE   4-PE   8-PE  16-PE  32-PE  64-PE 128-PE 256-PE
          0.855  0.749  0.696  0.687  0.665  0.654  0.642  0.632

Timing Information --------------------------------------------------
 Multilevel:             23.400
     Coarsening:                 12.460
     Initial Partition:           0.750
     Uncoarsening:                6.090
        Refinement:                       3.330
            Initialize:                           1.030
            Iterate:                              1.730
            Wrap Up:                              0.320
        Projection:                       2.720
     Splitting:                   2.770
 I/O:                     4.440
 Total:                  33.050
*******************************************************************
```

```
prompt% metis brack2.graph 120 100 4 11 1 1 2

*******************************************************************
 METIS 2.0   Copyright 1995, Regents of the University of Minnesota

Graph Information ---------------------------------------------------
   Graph: brack2.graph, Size: 62631,  733118, Parts: 120, Cto: 100
Options: HCM, BGR, GGP, Rec-Partition

Recursive Partitioning... -------------------------------------------
  120-way Edge-Cut:   42442, Balance: 1.002

Timing Information --------------------------------------------------
 Multilevel:             21.090
     Coarsening:                 13.450
     Initial Partition:           0.150
     Uncoarsening:                5.170
        Refinement:                       2.320
            Initialize:                           0.720
            Iterate:                              1.140
            Wrap Up:                              0.260
        Projection:                       2.700
        Balancing:                        0.130
     Splitting:                   2.270
 I/O:                     4.390
 Total:                  26.920
*******************************************************************
```

**Figure 5**: Output of `metis` for graph *brack2.graph*, for ordering and 128-way partition .




Graph File:

```
7 11
5 3 2
1 3 4
5 4 2 1
2 3 6 7
1 3 6
5 4 7
6 4
```

**(a) Unweighted Graph**

Graph File:

```
7 11 1
5 1 3 2 2 1
1 1 3 2 4 1
5 3 4 2 2 2 1 2
2 1 3 2 6 2 7 5
1 1 3 3 6 2
5 2 4 2 7 6
6 6 4 5
```

**(b) Weighted Graph Weights on edges**

Graph File:

```
7 11 11
4 5 1 3 2 2 1
2 1 1 3 2 4 1
5 5 3 4 2 2 2 1 2
3 2 1 3 2 6 2 7 5
1 1 1 3 3 6 2
6 5 2 4 2 7 6
2 6 6 4 5
```

**(c) Weighted Graph Weights both on vertices and edges**

**Figure 6**: Storage format for various type of graphs.

| | | | |
|---|---|---|---|
| 0000 | 0010 | 1000 | 1010 |
| 0001 | 0011 | 1001 | 1011 |
| 0100 | 0110 | 1100 | 1110 |
| 0101 | 0111 | 1101 | 1111 |

**Figure 7**: The number of the partitions for a 16-way partition.

implemented by METIS, namely `pmetis`, `kmetis`, and `ometis`.

The input to these routines is the adjacency structure of the graph and the weights of the vertices and edges (if any). In particular, a graph with $n$ vertices and $m$ edges is represented in METISlib by four arrays. These arrays are `xadj[n+1]`, `adjncy[m]`, `vwgts[n]`, and `ewgts[m]`. The arrays `xadj` and `adjncy` are used to store the adjacency structure of the graph, while the arrays `vwgts` and `ewgts`) are used to store the weights of the vertices and edges, respectively. If the graph is unweighted, the arrays `vwgts` and `ewgts` are `NULL`. The adjacency structure of the graph is stored as follows. Assuming that vertex numbering starts from 0 (C style), then the adjacency list of vertex $i$ is stored in the `xadj[i]...xadj[i+1]−1` section of array `adjncy` (*i.e.*, `adjncy[xadj[i]]` through and including `adjncy[xadj[i+1]−1]`). If the graph has weights on the edges, then the weight of edge `adjncy[j]` is stored in `ewgts[j]`, and if the graph has weights on the vertices, then `vwgts[i]` stores the weight of vertex $i$. METISlib also allows vertex numbering to start from 1 (Fortran style). In that case it is assumed that the arrays are passed from a Fortran program.

METISlib allocates all the memory it requires dynamically. This has the advantage that the user does not have to provide workspace, but if there is not enough memory, the routines in METISlib abort. Note that the routines in METISlib do not modify arrays that store the graph, but instead they make a private copy of them.

The calling sequence of the routines provides by METISlib and the meaning of the various parameters is as follows:

`PMETIS` (int *n, int *xadj, int *adjncy, int *vwgts, int *ewgts, int *weightflag, int *nparts

int *options, int *numbering, int *edgecut, int *partition)
KMETIS(int *n, int *xadj, int *adjncy, int *vwgts, int *ewgts, int *weightflag, int *nparts
int *options, int *numbering, int *edgecut, int *partition)
OMETIS(int *n, int *xadj, int *adjncy, int *options, int *numbering, int *perm, int *invperm)

**Parameters:**

**n** The number of vertices in the graph

**xadj, adjncy** The adjacency structure of the graph.

**vwgts, ewgts** The information about the weights of the vertices and edges.

**weightflag** Used to indicate if the graph is weighted. **weightflag** can take the following three values:

| | |
|---|---|
| 0 | No weights (vwgts and ewgts are NULL) |
| 1 | Weights on the edges only (vwgts = NULL) |
| 2 | Weights on the vertices only (ewgts = NULL) |
| 3 | Weights both on vertices and edges. |

**nparts** The number of parts to partition the graph.

**options** This is an array of 5 integers that is used to pass parameters to the various phases of the algorithm. If **options[0]=0**, then the default parameters are used (like the simple interface of METIS). If **options[0]=1**, then the remaining four elements of **options** are interpreted as follows (see Section 3.1 for the allowable values).

| | |
|---|---|
| options[1] | The number of vertices to coarsen down to (CoarsenTo) |
| options[2] | The matching type (MType) |
| options[3] | The initial partitioning algorithm (IPType) |
| options[4] | The refinement type (RType) |

In the case of **KMETIS options[3]** is unused.

**numbering** If **numbering=0**, then C-style numbering is assumed that starts from 0, if **numbering=1**, the Fortran-style numbering is assumed that starts from 1.

**edgecut** Upon successful completion, this variable stores the edge-cut of the partition.

**partition** This is a vector of size **n** and upon successful completion stores the partition vector of the graph.

**perm, invperm** These are vectors each of size **n** and upon successful completion they store the fill-reducing permutation and inverse-permutation. Note that the inverse permutation vector is the vector described in Section 5. Also, depending on the value of **numbering**, the permutation vectors start from 0 or 1.

Some examples of how to interface with the METISlib are provided in the `Examples` directory of METIS's distribution.

# 7 System Requirements and Contact Information

The distribution of METIS contains a number of files, that total to over 10,000 lines of code. It is written entirely in ANSI C, and is portable on most Unix systems that have an ANSI C compiler (the GNU C compiler will do). It has been extensively tested on AIX 3.2.5, SunOS 4.1 and 5.4 (Solaris 2.4), IRIX 5.3, and Unicos. Instructions on how to build and install METIS can be found in the file `INSTALL` of the distribution.

METIS minimizes the amount of memory it requires by dynamically allocating whatever memory it needs. It is hard to estimate the exact amount of memory required by METIS. For most graphs with $m$ edges, the amount of memory required is usually around $5m$ words. For instance, a 3D finite element mesh, with 250,000 vertices, will require around 60MBytes of memory, while a similar mesh with 1,000,000 vertices will require around 240MBytes of memory.

Even though, METIS contains no known bugs, it does not mean that all of its bugs have been found and fixed. If you find any problems, please send email to *karypis@cs.umn.edu*, with a brief description of the problem you have found. Also, any future updates to METIS will be made available on WWW at *http://www.cs.umn.edu/˜karypis/metis/metis.html*.

## References


[1] Stephen T. Barnard and Horst D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *Proceedings of the sixth SIAM conference on Parallel Processing for Scientific Computing*, pages 711–718, 1993.

[2] Earl R. Barnes. An algorithm for partitioning the nodes of a graph. *SIAM J. Algebraic Discrete Methods*, 3(4):541–550, December 1984.

[3] T. Bui and C. Jones. A heuristic for reducing fill in sparse matrix factorization. In *6th SIAM Conf. Parallel Processing for Scientific Computing*, pages 445–452, 1993.

[4] T. N. Bui, S. Chaudhuri, F. T. Leighton, and M. Sipser. Graph bisection algorithms with good average case behavior. *Combinatorica*, 7(2):171–191, 1987.

[5] Tony F. Chan, John R. Gilbert, and Shang-Hua Teng. Geometric spectral partitioning (draft). Technical Report In Preparation, 1994.

[6] Chung-Kuan Cheng and Yen-Chuen A. Wei. An improved two-way partitioning algorithm with stable performance. *IEEE Transactions on Computer Aided Design*, 10(12):1502–1511, December 1991.

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

[8] J. Garbers, H. J. Promel, and A. Steger. Finding clusters in VLSI circuits. In *Proceedings of IEEE International Conference on Computer Aided Design*, pages 520–523, 1990.

[9] A. George. Nested dissection of a regular finite-element mesh. *SIAM Journal on Numerical Ananlysis*, 10:345–363, 1973.

[10] A. George and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[11] John R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. In *Proceedings of International Parallel Processing Symposium*, 1995.

[12] T. Goehring and Y. Saad. Heuristic algorithms for automatic graph partitioning. Technical report, Department of Computer Science, University of Minnesota, Minneapolis, 1994.

[13] Anshul Gupta, George Karypis, and Vipin Kumar. Highly scalable parallel algorithms for sparse matrix factorization. Technical Report 94-63, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. Submitted for publication in *IEEE Transactions on Parallel and Distributed Computing*. Available on WWW at URL http://www.cs.umn.edu/~karypis/papers/sparse-cholesky.ps.

[14] Lars Hagen and Andrew Kahng. Fast spectral methods for ratio cut partitioning and clustering. In *Proceedings of IEEE International Conference on Computer Aided Design*, pages 10–13, 1991.

[15] Lars Hagen and Andrew Kahng. A new approach to effective circuit clustering. In *Proceedings of IEEE International Conference on Computer Aided Design*, pages 422–427, 1992.

[16] M. T. Heath and P. Raghavan. A Cartesian nested dissection algorithm. Technical Report UIUCDCS-R-92-1772, Department of Computer Science, University of Illinois, Urbana, IL 61801, 1992. To appear in *SIAM Journal on Matrix Analysis and Applications*, 1994.

[17] M. T. Heath and Padma Raghavan. A Cartesian parallel nested dissection algorithm. Technical Report 92-1772, Department of Computer Science, University of Illinois, Urbana, IL, 1992. To appear in *SIAM Journal on Matrix Analysis and Applications*, 1994.

[18] Bruce Hendrickson and Rober Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. Technical Report SAND92-1460, Sandia National Laboratories, 1992.

[19] Bruce Hendrickson and Rober Leland. The chaco user's guide, version 1.0. Technical Report SAND93-2339, Sandia National Laboratories, 1993.

[20] Bruce Hendrickson and Rober Leland. A multilevel algorithm for partitioning graphs. Technical Report SAND93-1301, Sandia National Laboratories, 1993.

[21] G. Karypis and V. Kumar. Analysis of multilevel graph partitioning. Technical Report TR 95-037, Department of Computer Science, University of Minnesota, 1995. Also available on WWW at URL http://www.cs.umn.edu/~karypis/papers/mlevel_analysis.ps.

[22] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report TR 95-035, Department of Computer Science, University of Minnesota, 1995. Also available on WWW at URL http://www.cs.umn.edu/~karypis/papers/mlevel_serial.ps. A short version appears in Intl. Conf. on Parallel Processing 1995.

[23] G. Karypis and V. Kumar. Multilevel *k*-way partitioning scheme for irregular graphs. Technical Report TR 95-XXX, Department of Computer Science, University of Minnesota, 1995. Also available on WWW at URL http://www.cs.umn.edu/~karypis/papers/mlevel_kway.ps.

[24] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 1970.

[25] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings Publishing Company, Redwood City, CA, 1994.

[26] J. W.-H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11:141–153, 1985.

[27] Gary L. Miller, Shang-Hua Teng, W. Thurston, and Stephen A. Vavasis. Automatic mesh partitioning. In A. George, John R. Gilbert, and J. W.-H. Liu, editors, *Sparse Matrix Computations: Graph Theory Issues and Algorithms*. (An IMA Workshop Volume). Springer-Verlag, New York, NY, 1993.

[28] Gary L. Miller, Shang-Hua Teng, and Stephen A. Vavasis. A unified geometric approach to graph separators. In *Proceedings of 31st Annual Symposium on Foundations of Computer Science*, pages 538–547, 1991.

[29] B. Nour-Omid, A. Raefsky, and G. Lyzenga. Solving finite element equations on concurrent computers. In A. K. Noor, editor, *American Soc. Mech. Eng*, pages 291–307, 1986.

[30] R. Ponnusamy, N. Mansour, A. Choudhary, and G. C. Fox. Graph contraction and physical optimization methods: a quality-cost tradeoff for mapping data on parallel computers. In *International Conference of Supercomputing*, 1993.

[31] Alex Pothen, H. D. Simon, Lie Wang, and Stephen T. Bernard. Towards a fast implementation of spectral nested dissection. In *Supercomputing '92 Proceedings*, pages 42–51, 1992.

[32] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis and Applications*, 11(3):430–452, 1990.

[33] P. Raghavan. Line and plane separators. Technical Report UIUCDCS-R-93-1794, Department of Computer Science, University of Illinois, Urbana, IL 61801, February 1993.

# A Characterization of Different Graph Partitioning Schemes

The graph partitioning problem is NP-complete. However, many algorithms have been developed that find a reasonably good partition. Spectral partitioning methods are known to produce good partitions for a wide class of problems, and they are used quite extensively [32, 31, 20]. However, these methods are very expensive since they require the computation of the eigenvector corresponding to the second smallest eigenvalue (Fiedler vector). Execution of the spectral methods can be speeded up if computation of the Fiedler vector is done by using a multilevel algorithm [1]. This multilevel spectral bisection algorithm (MSB) usually manages to speedup the spectral partitioning methods by an order of magnitude without any loss in the quality of the edge-cut. However, even MSB can take a large amount of time. In particular, in parallel direct solvers, the time for computing ordering using MSB can be several orders of magnitude higher than the time taken by the parallel factorization algorithm, and thus ordering time can dominate the overall time to solve the problem [13].

Another class of graph partitioning techniques uses the geometric information of the graph to find a good partition. Geometric partitioning algorithms [16, 17, 33, 28, 27, 29] tend to be fast but often yield partitions that are worse than those obtained by spectral methods. Among the most prominent of these scheme is the algorithm described in [28, 27]. This algorithm produces partitions that are provably within the bounds that exist for some special classes of graphs (that includes graphs arising in finite element applications). However, due to the randomized nature of these algorithms, multiple trials are often required (5 to 50) to obtain solutions that are comparable in quality to spectral methods. Multiple trials do increase the time [11], but the overall runtime is still substantially lower than the time required by the spectral methods. Geometric graph partitioning algorithms are applicable only if coordinates are available for the vertices of the graph. In many problem areas (*e.g.*, linear programming, VLSI), there is no geometry associated with the graph. Recently, an algorithm has been proposed to compute coordinates for graph vertices [5] by using spectral methods. But these methods are much more expensive and dominate the overall time taken by the graph partitioning algorithm.

Another class of graph partitioning algorithms reduce the size of the graph (*i.e.*, coarsen the graph) by collapsing vertices and edges, partition the smaller graph, and then uncoarsen it to construct a partition for the original graph. These are called multilevel graph partitioning schemes [3, 6, 14, 15, 20, 8, 30]. Some researchers investigated multilevel schemes primarily to decrease the partitioning time, at the cost of somewhat worse partition quality [30]. Recently, a number of multilevel algorithms have been proposed [3, 20, 6, 15, 8, 22] that further refine the partition during the uncoarsening phase. These schemes tend to give good partitions at a reasonable cost. In particular, in [22] we showed that multilevel schemes can provide better partitions than spectral methods at a much lower cost for a variety of graphs arising in various domains including finite element methods, linear programming, VLSI, and transportation. In [22] we investigated the effectiveness of many different choices for all three phases: coarsening, partition of the coarsest graph, and refinement. In particular, we presented a new coarsening heuristic (called heavy-edge heuristic) for which the size of the partition of the coarse graph is within a small factor of the size of the final partition obtained after multilevel refinement [21]. We also present a new scheme for refining during uncoarsening that is much faster than the Kernighan-Lin refinement used in [20]. Our experiments show that our scheme consistently produces partitions that are better than those produced by spectral partitioning schemes in substantially smaller timer. We also used our graph partitioning scheme to compute fill reducing orderings for sparse matrices. Surprisingly, our scheme substantially outperforms the multiple minimum degree algorithm [26], which is the most commonly used method for computing fill reducing orderings of a sparse matrix.

The various graph partitioning schemes that have been developed differ widely in the edge-cut quality produced, run time, degree of parallelism, and applicability to certain kind of graphs. Often, it is not clear as to which scheme is better under what scenarios. In Table 1 we show three different variations of spectral partitioning [32, 31, 20, 1], the multilevel partitioning scheme described in [22] and implemented in METIS, the levelized nested dissection [9], the Kernighan-Lin partition [24], the coordinate nested dissection (CND) [16], two variations of the inertial partition [29, 19], and two variant of geometric partitioning [28, 27, 11].

| | Number of Trials | Needs Coordinates | Quality | Local View | Global View | Run Time | Degree of Parallelism |
|---|---|---|---|---|---|---|---|
| Spectral Bisection | 1 | no | ●●●● | ○ | ●●●● | ■■■■ | ▲▲ |
| Multilevel Spectral Bisection | 1 | no | ●●●● | ○ | ●●●● | ■■■ | ▲▲ |
| Mulitlevel Spectral Bisection-KL | 1 | no | ●●●●●● | ●● | ●●●● | ■■■ | ▲▲ |
| Multilevel Partitioning | 1 | no | ●●●●●● | ●● | ●●●● | ■■ | ▲▲ |
| Levelized Nested Dissection | 1 | no | ●● | ○ | ●● | ■■ | ▲▲ |
| | 1 | no | ●● | ●● | ○ | ■■ | ▲ |
| Kernighan-Lin | 10 | no | ●●●◐ | ●● | ●◐ | ■■■ | ▲▲ |
| | 50 | no | ●●●● | ●● | ●● | ■■■■◧ | ▲▲ |
| Coordinate Nested Dissection | 1 | yes | ● | ○ | ● | ■ | ▲▲▲ |
| Inertial | 1 | yes | ●● | ○ | ●● | ■ | ▲▲▲ |
| Inertial-KL | 1 | yes | ●●●● | ●● | ●● | ■■ | ▲ |
| | 1 | yes | ●● | ○ | ●● | ■ | ▲▲▲ |
| Geometric Partitioning | 10 | yes | ●●●◐ | ○ | ●●●◐ | ■■ | ▲▲▲ |
| | 50 | yes | ●●●● | ○ | ●●●● | ■■◧ | ▲▲▲ |
| | 1 | yes | ●●●● | ●● | ●● | ■■ | ▲ |
| Geometric Partitioning-KL | 10 | yes | ●●●●●◐ | ●● | ●●●◐ | ■■■ | ▲▲ |
| | 50 | yes | ●●●●●● | ●● | ●●●● | ■■■■◧ | ▲▲ |

**Table 1**: Characteristics of various graph partitioning algorithms.

Table 1 for each graph shows a number of characteristics. The first column shows the number of trials that are often performed for each partitioning algorithm. For example, for Kernighan-Lin different trials can be performed each starting with a random partition of the graph. Each trial is a different run of the partitioning algorithm, and the overall partition is determined as the best of these multiple trials. As we can see from this table, some algorithms require only a single trial either because, multiple trials will give the same partition (*i.e.*, the algorithm is deterministic), or the single trial gives very good results (as in the case of multilevel graph partitioning). However, for some schemes like Kernighan-Lin and geometric partitioning, different trials yield significantly different edge-cuts; hence, these schemes usually require multiple trials in order to produce good quality partitions. For multiple trials, we only show the case of 10 and 50 trials, as often the quality saturates beyond 50 trials, or the run time becomes too large. The second column shows whether the partitioning algorithm requires coordinates for the vertices of the graph. Some algorithms such as CND and Inertial can work only if coordinate information is available. Others only require the set of vertices and edges connecting them.

The third column of Table 1 shows the relative quality of the partitions produced by the various schemes. Each additional circle corresponds to roughly 10% improvement in the edge-cut. The edge-cut quality for CND serves as the base, and it is shown with one circle. Schemes with two circles for quality should find partitions that are roughly 10% better than CND. This column shows that the quality of the partitions produced by our multilevel graph partitioning algorithm and the multilevel spectral bisection with Kernighan-Lin is very good. The quality of geometric partitioning with Kernighan-Lin refinement is also equally good, when around 50 or more trials are performed[1]. The quality of the other schemes is worse than the above three by various degrees. Note that for both Kernighan-Lin partitioning and geometric partitioning the quality improves as the number of trials increases.

The reason for the differences in the quality of the various schemes can be understood if we consider the degree of quality as a sum of two quantities that we refer to as ***local view*** and ***global view***. A graph partitioning algorithm has a local view of the graph if it is able to do localized refinement. According to this definition, all the graph partitioning algorithms that use at various stages of their execution variations of the Kernighan-Lin partitioning algorithm possess this local view, whereas the other graph partitioning algorithms do not. Global view refers to the extent that the graph partitioning algorithm takes into account the structure of the graph. For instance, spectral bisection algorithms take into account only global information of the graph by minimizing the edge-cut in the continues approximation of the discrete problem. On the other hand, schemes such as a single trial of Kernighan-Lin, utilize no graph structural information, since it starts from a random bisection. Schemes that require multiple trials, improve the amount of global graph structure they exploit as the number of trials increases. Note that the sum of circles for global and local view columns is equal to the number of circles for quality for various algorithms. The global view of multilevel graph partitioning is among the highest of that of the other schemes. This is because the multilevel graph partitioning captures global graph structure at two different levels. First, it captures global structure through the precess of coarsening [21], and second, it captures global structure during the initial graph partitioning by performing multiple trials.

The sixth column of Table 1 shows the relative time required by different graph partitioning schemes. CND, inertial, and geometric partitioning with one trial require relatively small amount of time. We show the run time of these schemes by one square. Each additional square corresponds to roughly a factor of 10 increase in the run time. As we can see, spectral graph partition schemes require several orders of magnitude more time than the faster schemes. However, the quality of the partitions produced by the faster schemes is relatively poor. The quality of the geometric partitioning scheme can be improved by increasing the number of trials and/or by using the Kernighan-Lin algorithm, both of which significantly increase the run time of this scheme. On the other hand, multilevel graph partitioning requires moderate amount of time, and produces partitions of very high quality.

[1] This conclusion is an extrapolation of the results presented in [11] where it was shown that the geometric partitioning with 30 trials (*default geometric*) produces partitions comparable to that of multilevel spectral bisection without Kernighan-Lin refinement.