# An Implementation of State-of-the-Art Graph Bisection Algorithms *

*Nan Dun*
*Department of Computer Science, the University of Tokyo*

April 28, 2006

**Abstract**

In recent decades, various graph partitioning heuristic algorithms have been developed by researchers, which are widely used in "real-world" applications. As a starting point of graph partition problem, graph bisection algorithms provide basic approaches and solutions for other graph partition instances. This work implemented part of contemporary graph bisection algorithms which produce high-quality partitions for graphs in wide-range domains. These algorithms include Kernighan-Lin heuristic[3], Linear Kernighan-Lin heuristic[4], Greedy Growing Partitioning[5], Min-Max Greedy heuristic[6], Randomized Tabu Search heuristic[6], Reactive Randomized Tabu Search heuristic[6]. Multilevel graph partition technique[5] has not been included in current implementation to achieve better performance. Instead, these heuristics are extended in grid environment so as to significantly reduce run-time by distributing mutually exclusive iterations over computing nodes with support from GXP cluster shell[9]. Further, this parallel bisection needs no communication among computing nodes, thus bearing a high fault-tolerance. Finally, experiments show that although with more computation effort comparing to mulitilevel partition, this graph bisection library is still practically useful with its state-of-the-art result.

**Keywords**: Graph bisection, graph partitioning, heuristic algorithms, Kernighan-Lin, greedy heuristic, Tabu search, GXP cluster shell.

*Grid Challenge 2006, Japan. http://www.hpcc.jp/sacsis/2006/grid-challenge/

# 1 Introduction

In mathematics, the graph ***k-partition*** problem is defined as follows: Given a *unweighted* graph $G = (V, E)$, partition $V$ into $k$ subsets, $V_1, V_2, \ldots, V_k$ such that $V_i \cap V_j = \phi$ for $i \neq j$, $|V_i| = |V|/k$, and $\bigcup V_i = V$, and the number of edges of $E$ whose incident vertices belong to different subsets, or ***edge-cut*** is minimized. And a graph *2-partition*, ***bipartition*** or ***bisection*** problem can be summarized as: Given a unweighted graph $G = (V, E)$, partition $V$ into $V_1$ and $V_2$ such that $V_1 \cup V_2 = V$, $V_1 \cap V_2 = \phi$ and $||V_1| - |V_2|| \leq 1$ (0 if $|V|$ is even, 1 otherwise), meanwhile, the edge-cut $|\{E(v_i, v_j)_{v_i \in V_1, v_j \in V_2} \in E\}|$ is minimized. Note that weighted graph partitioning can be inherently extended from unweighted graph partitioning.

Graph partition problem is a classical problem and has been extensively investigated over many years. Graph k-partition problem is NP-complete[1], as well as graph bisection problem[2]. Finding approximate solution is also NP-hard, loosing the restriction of equal-size sets can lead avaiable but not practical algorithms. Most of solutions of graphs in wide range are produced by heuristics.

The importance of graph partition not only comes from mathematics, such as sparse matrix-vector multiplication and Gaussian elimination, but also from practical applications, including load balancing, mesh distributing, VLSI and large network design. Therefore, this implementation of contemporary bisection algorithms is trying to provide practical usability for real-world problems rather than demonstration only.

# 2 Graph Bisection Algorithms

Many graph partitioning heuristic algorithms have been developed by researchers recently. There are three classes of graph partitioning techniques[5]: Spectral partition, geometric partition and multilevel partition. Spectral partitioning is good at producing better result but usually costing more execution time. Geometric partitioning algorithms uses the geometric information of the graph to find a good partition, it is faster but yields worse result than spectral techniques. Multilevel schemes are usually applied for very large graphs (100,000-500,000 vertices) to obtain a short execution time. The multilevel partitioning significantly reduce the size of graph by collapsing vertices and edges, partition the smaller graph, and then uncoarsen it to construct a partition for the original graph. The combination of multilevel and other techniques is main stream of nowaday graph partition. Since our work starts from dealing with relative small graphs (<100,000 vertices), multilevel techniques is not included. In this section, the heuristc algorithms are discussed in consequent as well as time sequence, which gradually approach better results and performance.

## 2.1 Kernighan-Lin Heuristic

Kernighan-Lin heuristic[3] was originally developed to solve telephone network problem and now becomes a classical one. KL heuristic holds a $O(|V|^3)$, or better $O(|E|\log|E|)$ complexity. Before getting into the algorithm given below as Figure 1, a few concepts should be explained first.

- ***weight***: Denote connection degree of two vertices $v_i$ and $v_j$. For unweighted graphs, it is 1 if there is a edge between $v_i$ and $v_j$, 0 otherwise.

$$\omega(v_i, v_j)_{v_i,\ v_j \in V} = \begin{cases} 0 & E(v_i, v_j) \notin E \\ 1 & E(v_i, v_j) \in E \end{cases} \tag{1}$$

- ***degree of vertex***: Degree of vertex $v$ is the number of edges one of whose endpoint is $v$.

$$Degree(v) = |\{E(v, v^{'})_{v^{'} \in V} \in E\}| = \sum_{v^{'} \in V} \omega(v, v^{'}) \tag{2}$$

- ***internal cost***: The number of internal edges (edge whose endpoints lie in same set).

$$Internal(v_i)_{v_i \in V_1} = \sum_{v_j \in V_1} \omega(v_i, v_j) \tag{3}$$

- ***external cost***: The number of internal edges (edge whose endpoints lie in same set).

$$External(v_i)_{v_i \in V_1} = \sum_{v_j \in V_2} \omega(v_i, v_j) \tag{4}$$

- ***gain***: The variance of edgcut after moving a vertex from one set to another.

$$Gain(v_i)_{v_i \in V} = Internal(v_i) - External(v_i) \tag{5}$$

- ***cut***: From (1)-(4), it is easy to derive the variance of edgcut after swapping two vertices in two different set.

$$Cut(v_i, v_j)_{v_i \in V_1, v_j \in V_2} = 2\omega(v_i, v_j) + Gain(v_i) + Gain(v_j) \tag{6}$$

Kernighan-Lin algorithm starts from a initial bisection, since its basic idea is to swap two vertices in different sets if this exchange can bring less *edgecut* comparing to former ones. The initial bisection can be obtained simply by half-half dividing (first half of vertices form a set, and second half of vertices form another), or by random bisection (randomly choose half of

vertices as one set, and the rest forms another). KL heuristic is sensitive to this initial configuration, thus, also as experiments show, random bisection tends to be generate better result than half-half bisection. Due to this, KL heuristic is usually used as refinement of bisection produced by other heuristics, and we will discuss this in later sections.

KERNIGHAN-LIN

```
1    forall vertex ∈ V
2        Lock[vertex] ← unlock
3    edgecut_min ← edgecut_init, Cut[0] ← edgecut_init
4    bestchange ← 0
5    for t ← 1 to |V|/2
6        Cut[t] ← ∞
7        forall vertex_i ∈ {v ∈ setL|Lock[v] = unlock}
8            forall vertex_j ∈ {v ∈ setR|Lock[v] = unlock}
9                cut ← 2ω(vertex_i, vertex_j)+Gain(vertex_i)+Gain(vertex_j)
10               if cut < Cut[t] then
11                   Pair[t] ← (vertex_i, vertexj)
12                   Cut[t] ← cut
13               (min_i, min_j) ← Pair[t]
14               Lock[min_i] ← lock, Lock[min_j] ← lock
15               forall vertex ∈ {v ∈ V|Lock[v] = unlock}
16                   if vertex ∈ setL then
17                       Gain(vertex) ← Gain(vertex) + 2ω(vertex, min_j)
                                                     −2ω(vertex, min_i)
18                   else
19                       Gain(vertex) ← Gain(vertex) + 2ω(vertex, min_i)
                                                     −2ω(vertex, min_j)
20       Cut[t] ← Cut[t − 1] + Cut[t]
21       if Cut[t] < edgecut_min
22           bestchange ← t
23           edgecut_min ← Cut[t]
24   for i ← 1 to bestchange
25       swap Pair[i]
```

**Figure 1 Kernighan-Lin Heuristic**

Given an initial bisection, KL first mark all vertices *unlock* (line 1-2), which means these vertices are free to move. Since we swap two vertices, we only need iterate for $|V|/2$ times (line 5). By checking through each pair which crossing two sets (line 7-8), we search for a pair such that it cause greatest decrease of *edge-cut* (line 10-13). Then we lock this pair as if they were swapped (line 15), and update gains of rest free vertices (line 17-19). At the end of each iteration, we accumulate former swapping sequence and record the time when minimum edgecut was reached (line 20-23). Finally, real swappings are performed to generate new bisection (line 24-25).

From algorithm above, we can find that KL heuristic may escape "local minimum" where switching no pair helps. Usually, for very small graphs, the *edge-cut* will converge after 2-4 runs. And for random graphs, the probability of convergence in a single run appears to be $2^{-|V|/30}$. Thus KL is relative much slower comparing to following algorithms we will discuss.

### 2.2 Linear Kernighan-Lin Heuristic

Linear Kernighan-Lin heuristic[4] is one of KL's successors. The essential algorithm of LKL is similar to Figure 1, but achieves $O(|E|)$ complexity instead of KL's $O(|V|^3)$ due to using proper data structures during implementation. Thus we delay the discussion of LKL heuristic to Section 3.2.

### 2.3 Greedy Growing Partitioning

The graph growing algorithm comes from the intuition that bisection starts from a seed and grow a partition around it in a breath-first-search way, and stop until half of vertices have been included. Thus the quality of bisection also dependes on this seed. Its complexity is $O(|E|^2)$. Graph growing heuristic is usually used as starting a new bisection and pass the result to other heuristic, e.g. KL heuristic, for refining.

- ***adjacent vertices***: The adjacent vertices of a vertex $v$ is those vertices sharing edges with $v$.

$$adjacent(v) = \{v^{'} \in V | E(v, v^{'})_{v \in V} \in E\} \tag{7}$$

- ***boundary of set***: Given a partition $V_1$ and $V_2$ of graph $V$. A subset of $V_1$ such that each vertices in it has at least one neighbour in $V_2$ is called *boundary* of $V_1$.

$$boundary(V_1) = \{v \in V_1 | \exists E(v, v^{'})_{v^{'} \in V_2} \in E\} \tag{8}$$

- ***frontier of set***: Given a partition $V_1$ and $V_2$ of graph $V$. A subset of $V_2$ such that all vertices in it have at least one neighbour in $V_1$ is called *frontier* of $V_1$.

$$frontier(V_1) = \{v \in V_2 | \exists E(v, v^{'})_{v^{'} \in V_1} \in E\} \tag{9}$$

Greedy growing algorithm extends graph growing heuristic by replacing the breath-first-search with greedily including the vertex which causes largest decrease (or smallest increase) to the current partition. As in KL, for each vertex we define *gain* in the *edge-cut* by moving this vertex from one set to another. Additional effort is to sort the vertices of current set's (partition) *frontier* by their *gains* in non-decreasing order, and then the vertex with

smallest *gain* is inserted. After a vertex is added, gains of its *adjacent vertices* already in the frontier are updated, those *adjacent vertices* not in frontier are inserted to frontier.

```
GREEDYGROWING
1    setL ← φ, setR ← V
2    seed ← random vertex ∈ setR
3    setL ← setL ∪ {seed}, setR ← setR \ {seed}
4    edgecut ← 0
5    setTobeadded ← {v ∈ setR|E(seed, v) ∈ E}
6    while |setL| < |V|/2
7        if setTobeadded = φ then
8            vertex ← random vertex ∈ setR
9            setTobeadded ← setTobeadded ∪ {vertex}
10       gain_min ← min_{v∈setTobeadded} Gain(v)
11       vertex_min ← first vertex ∈ {v ∈ setTobeadded|Gain(v) = gain_min}
12       setTobeadded ← setTobeadded\{vertex_min}
13       setL ← setL ∪ {vertex_min}, setR ← setR\{vertex_min}
14       edgecut ← edgecut + gain_min
15       if adjacent(vertex_min) ≠ φ then
16           forall vertex ∈ adjacent(vertex_min)
17               if vertex ∈ frontier(setTobeadded) then
18                   update Gain(vertex)
19               else
20                   setTobeadded ← setTobeadded ∪ {vertex}
```

**Figure 2 Greedy Growing Heuristic**

Greedy growing algorithm is also sensitive to initial seed, but less sensitive than graph growing region. A typical usage is to perform indenpendent runs of greedy growing algorithm with different seed, and select the bisection with smallest *edge-cut*.

## 2.4 Min-Max Greedy Partitioning

Min-Max Greedy heuristic is also used as produce an initial bisection. It can be similarily considered as the "twin" version of greedy growing algorithm, since it starts from two seed, and growing their own regions by adding candidate vertices in turn. Min-Max Greedy appends additional criteria comparing to Greedy Growing heuristic when some vertex is going to be added, based on following definition.

- ***connection from vertex to set***: The number of edges whose one endpoint is $v$ and another endpoint is in $V'$ is called the *connection*

from $v$ to $V'$.

$$Conn(v, set) = |\{E(v, v')_{v' \in set} \in E\}| \tag{10}$$

From Greedy Growing heuristic, we choose a candidate which has minimum *gain*. Recalling (5), to minimize the *gain*, one can minimize the *internal cost* and maximize the *external cost*. Thus, Min-Max Greedy choose a candidate by minimizing its *connection* to other set, or edgecut, and maximizing its *connection* to add set.

There are two alternatives to screening the candidates. One, denoted as FirstMin, is firstly choosing a group of candidates which have maximum *connection* to add set, and then select the first one with minimum *connection* to otherset among them. Another, FirstMax, select the first vertex with maximum *connection* to a add set from the candidates set satisfying minimum connection to other set. Since at the beginning, the number of vertices that are connected to a given set by one or more edges is very small, FirstMax is more efficient than FirstMin [6].

```
MinMaxGreedy
1   setL ← φ, setR ← φ
2   seedL ← random vertex ∈ {1, ..., n}
3   seedR ← random vertex ∈ {1, ..., n} \ {seedL}
4   setTobeadded ← V \ {seedL, seedR}
5   edgecut ← 0
6   if (seedL, seedR) ∈ E then edgecut ← 1
7   setAdd ← setR, setOther ← setL
8   while |setTobeadded| > 0
9       setAdd ↔ setOther
10      minedges ← min_{v∈setTobeadded} Conn(v, setOther)
11      bestvertex ← FirstMax(setAdd, setOther)
12      setAdd ← setAdd ∪ {bestvertex}
13      edgecut ← edgecut + minedges
14      setTobeadded ← setTobeadded \ {bestvertex}
```

**Figure 3 Min-Max Heuristic**

```
FirstMax(setAdd, setOther)
15  min ← min_{v∈setTobeadded} Conn(v, setOther)
16  candidates ← {v ∈ setTobeadded | Conn(v, setOther) = min}
17  max ← max_{v∈candidates} Conn(v, setAdd)
18  bestvertex ← first vertex ∈ {v ∈ candidates | Conn(v, setAdd) = max}
19  return bestvertex
```

**Figure 4 First-Max Routine**

## 2.5 Randomized Tabu Search Heuristic

Tabu, or prohibition-based search can be considered as a variant of basic local-search scheme, such as KL heuristic. Its purpose is to optimize local optimality. [6] gives the relationship between Kernighan-Lin heuristic and Tabu Search. Later we will see under some conditions, KL is equivalent to Tabu. Recalling KL algorithm, when a pair of vertices has been swapped, they are locked (prohibited) for further moving. This scheme suggests that for one single pass of KL, the result will stay at local optimization, even better configuration can be achieved by moving those vertices that have been swapped before. But for Tabu Searching, after a pair of vertices swapped, instead of being locked forever (in this pass), they are locked for a period of time. Then they are avaiable to move again when time exceed this period.

Tabu length refers to this period of time, which is usually a fraction of vertices number, line 1. And a *LastUsed* list records the time when some vertex is moved (line 6). In *BestMove* routine, a vertex is legal to move if its last-move time is not in prohibition period (line 11). All Tabu search will do $|V|$ iterations, and when the bisection is balanced, the minimum *edgecut* and corresponding configuration are recorded (line 8). Actually, if *LastUsed* time for each vertex is set to $-\infty$ and tabu length is set to more than $|V|/2$, Tabu Search becomes equivalent to Kernighan-Lin.

```
FIXEDTABUSEARCH(frac_Tabu, iterations)
1    Tabu ← ⌊frac_Tabu × n⌋
2    for i ← 1 to iterations
3        if |setL| ≥ n/2 then setAdd ← setR, setOther ← setL
                         else setAdd ← setL, setOther ← setR
4        bestvertex ← BESTMOVE(setOther)
5        setAdd ← setAdd ∪ {bestvertex}, setOther ← setOther \ {bestvertex}
6        LastUsed[bestvertex] ← time
7        time ← time + 1
8        if |setL − setR| ≤ 1 and edgecut < edgecut_min then
             edgecut_min ← edgecut
```

**Figure 5 Fixed Tabu Search Heuristic**

```
BESTMOVE(set)
9    for gain ← max_{i∈set} Gain(i) downto min_{i∈set} Gain(i)
10       forall vertex ∈ {j ∈ set such that Gain(j) = gain}
11           if LastUsed[vertex] < (time − Tabu) return vertex
```

**Figure 6 BestMove Routine**

Randomized Tabu Search is simple extension of Fixed Tabu Search. RTS passes the initial bisection generated by Min-Max Greedy (line 14) to Fixed

Tabu Search with randomly choosed tabu length. Different tabu lengthes are tried and the best bisection met during these searches are recoreded.

Experiements[6] shows that the tabu fractions ranging from 0.01 to 0.25 are sufficient to approach best result, more fractions will not improve the result.

```
RANDOMIZEDTABUSEARCH(iterations, individual)
12   time ← 0
13   while time < iterations
14       MINMAXGREEDY
15       time_end ← time + individual
16       while time < time_end
17           frac_Tabu ← random value ∈ {0.01, 0.02, ..., 0.25}
18           FIXEDTABUSEARCH(frac_Tabu, |V|)
```

**Figure 7 Randomized Tabu Search Heuristic**

## 2.6 Reactive Randomized Tabu Search Heuristic

Randomized Tabu Search performs local-search with statically given tabu fraction. And Reactive Randomized Tabu Search is in a self-tuning way to dynamically choose tabu length. Parameters, especially the tabu length, are dynamically optimized depending on the statistical characteristics of current graph but not specific category of graphs or certain local-optimized configuration.

The RRTS algorithm (Figure 9) starts from looking for a best tabu fraction by a $\widehat{\text{S}}$coring routine (Figure 8). $\widehat{\text{S}}$coring routine tries all given tabu fraction (line 2) and evaluate their results (line 14) based on *edge-cut* decrease during this pass. All votes are normalized from 0.1 to 1 (line 15-17). At last, the fraction that gains highest vote is returned as the best fraction (line 18-19). Further, the best configuration met in individual trial is recorded. (line 13)

```
SCORING(trials)
1    elite ← φ
2    forall frac_Tabu ∈ {0.01, 0.02, ..., 0.25}
3        vote[frac_Tabu] ← 0
4        for i ← 1 to trials
5            MINMAXGREEDY
6            LOCALSEARCH
7            edgecut_min ← edgecut, edgecut_start ← edgecut
8            time_start ← time
9            do
```

```
10              FIXEDTABUSEARCH(frac_Tabu, 2⌊frac_Tabu × |V|⌋ + 1)
11              LOCALSEARCH
12          while(time < time_start + |V|/2)
13          elite ← elite ∪ {best bisection with frac_Tabu}
14          vote[frac_Tabu] ← vote[frac_Tabu] + (edgecut_start − edgecut_min)/(time − time_start)
15  forall frac_Tabu ∈ {0.01, 0.02, ..., 0.25}
16      if max_vote ≠ min_vote then
17          vote[frac_Tabu] ← 0.1 + 0.9 × (vote[frac_Tabu] − min_vote)/(max_vote − min_vote)
18  bestfrac_Tabu ← smallest frac_Tabu such that vote[frac_Tabu] = max_vote
19  return bestfrac_Tabu
```

**Figure 8 Scoring Rountine**

The RRTS algorithm does 3 trials of *Scoring*, larger number of trials will not increase the performance[6]. The best configuration and tabu fraction are extracted after *Scoring*, or intial a bisection by Min-Max Greedy Routine. In RRTS, a wider range search is performed (see line 12 and 31). If the best *edge-cut* is not found in *Scoring* trials, a random tabu fraction is selected with probability proportional to their votes (line 33-34).

```
REACTIVERANDOMIZEDTABUSEARCH(iterations, individual)
20  bestfrac_Tabu ← SCORING(3)
21  for i ←  to ⌈iterations/individual⌉
22      if elite ≠ φ then extract bisection from elite
23                   else MINMAXGREEDY
24      frac_Tabu ← bestfrac_Tabu
25      t_individual ← time
26      do
27          t_start ← time
28          do
29              FIXEDTABUSEARCH(frac_Tabu, 2⌊frac_Tabu × |V|⌋ + 1)
30              LOCALSEARCH
31          while(time < time_start + |V|)
32          if time_min ≤ time_start then
33              frac_Tabu ← random frac_Tabu with probability ∝ vote[frac_Tabu]
34      while(time < time_individual + individual)
```

**Figure 9 Reactive Randomized Tabu Search**

# 3 Graph bisection library

Our graph bisection library implemented all heuristic algorithms discussed above. The implementation consists of about 3,000 lines C code. The library

provides simplicity APIs and clear interface for tunning heuristics. In this section, we firstly introduce the usage of this library, and then we give a detailed description of its implementation.

All source code and related materials are available at author's website: http://www.yl.is.s.u-tokyo.ac.jp/∼dunnan/graphbisection/.

## 3.1 Basic Guide

### 3.1.1 Installation

Follow the instructions to build the graph bisection library.

1. Unpack the tarball
   `tar -zxf gb.tar.gz`
   or
   `gunzip -c gb.tar.gz | tar xf -`

2. Build and intall
   `./INSTALL`

### 3.1.2 Using APIs

Before APIs, we present the essential data structure used over all bisection routines.

```
 1   typedef int integer;
2   struct GB_GRAPH {
3      char filename[GB_MAX_FILENAME]; /* graph's filename */
4      integer nv;          /* Number of vertices */
5      integer ne;          /* Number of edges */
6      integer **adjlist;   /* Vertices adjacent list */
7      int vparity;         /* Parity of vertices number */
8      integer maxvdeg;     /* Maximum vertex degree */
9      integer minvdeg;     /* Minimum vertex degree */
10     integer *belonglist; /* Vertices belonging list */
11     integer edgecut;     /* Edge-cut */
12  };
```

**Figure 10 GB_GRAPH Data Structure**

First, we use "int" type as vertex's index type. In Linux `INT_MAX` (2147483647 by ANSI) is pretty enough for most graphs, however, in Windows `INT_MAX` (32767) may be overflowed easily, thus you can define "long" type instead. Note that using "unsigned" type is at risk since it may cause unpredicatable result. Note that the index of vertices starts from 1, not 0.

`GB_GRAPH.adjlist` is a 2-dimentional integer array. the vertex $v$'s adjacent vertices list is given by `adjlist[v]`, in which `adjlist[v][0]` indicates the number of adjacent vertices of $v$, the adjacent vertices start from `adjlist[v][1]`.

GB_GRAPH.vparity indicates the parity of vertices number, which might be GB_NUM_EVEN or GB_NUM_ODD. GB_GRAPH.maxdeg and GB_GRAPH.minvdeg record the maximum and minimum vertex degree in graph, they are used to improve the search performance in implementation.

GB_GRAPH.belonglist indicates which partition the vertex belongs to, GB_PAR_L (left partition), GB_PAR_R (right partition) or GB_PAR_N (no partition). GB_GRAPH.edgecut is the *edge-cut* of bisection.

Now we illustrate APIs usage by following sample program.

```
 13   #include <stdlib.h>
14   #include <stdio.h>
15   #include "gb.h"
16
17   int main(int argc, char *argv[]){
18      GB_GRAPH mygraph;
19      char *file;
20      int hostid, hostnum, seed[2], iter;
21      doulbe timeout; integer threshold;
22
23      /* argument check */
24      file = argv[1];
25      hostid = atoi(argv[2]);
26      hostnum = atoi(argv[3]);
27      iter = atoi(argv[4]);
28      timeout = atoi(argv[5]);
29
30      GB_Initial(file, &mygraph);
31      GB_SeedDistribute(hostid, hostnum, seed);
32      GB_Bisect(&mygraph, GB_KL, seed, iter, timeout, threshold);
33      GB_Check(&mygraph);
34      GB_Finalize(&mygraph);
35
36      return 0;
37   }
```

**Figure 11 Sample Program**

- `GB_Initial(char *filename, GB_GRAPH *graph)`
  Allocate space for `graph` structure, and fill it with content read from `filename`.
- `GB_SeedDistribute(char *filename, GB_GRAPH *graph)`
  Divide the whole seed range into `hostnum` sections and assign them based on `hostid`. For single host running, simply set `hostid` and `hostnum` to 1. This routine is especially for GXP usage and will be discussed later.
- `GB_Bisect(GB_GRAPH *graph, GB_ALGO options, int *seed, int iter, double timeout, integer threshold)`
  Bisect given `graph` by heuristic indicated by `options`, which could be `GB_KL, GB_LKL, GB_GG, GB_MMG, GB_RTS` or `GB_RRTS`. `iter` shows how many independent iterations will be performed, set to -1 means infinite

iterations. And partitioning procedure will return when elapsed time exceeds `timeout` (in seconds), or *edge-cut* less/equal than `threshold`, or `iter` iterations is reached. The best bisection will be passed out as `GB_GRAPH.belonglist` when this routine returns.

- `GB_Check(GB_GRAPH *graph)`
  Check graph and bisection's integrity and print out check information.
- `GB_Finialize(GB_GRAPH *graph)`
  Free space for `graph` structure.

### 3.1.3 Running Program

#### Single Host

Running single program is trivial.

```
1   ./main file hostid hostnum iter timeout threshold
```

#### Cluster Environment

First, a better bisection is approached by independent iterations of one or more heuristics passes. In general case, more iteration leads to better result. Thus, it is naturally to distribute these independent iterations over hundreds of compute nodes within cluster. For 100 compute nodes, we can let each node run a 1-iteration approximately 100 iterations in total, or run 100-iterations on each node as approximate 10,000 iterations in total.

Second, following above's scheme, one important problem is how to keep these iteration as "independent" as possible, since there may have duplicate initial configurations which end with same result, which is only time wasting. Recalling some heuristics starting from random seeds (Greedy Growing starts from 1 seed, Min-Max Greedy starts from 2 seeds), they are sensitive to initial configuration, or seeds. And another important phenomenon in graph bisetion problem is the casual relationship between initial configuration to specific result (mostly "best" result), which means one will never go to the "best" result until the heuristic get on the "right" path. A proof of this barrier is that our experiements on 500 compute nodes solving some large-scale graph ($\geq$50,000 vertices), only 1 or 2 node will finally get the "best" result, and the rest will stay at some average good bisection but still has a distance to the "best" result. Of course, this also shows that in NP-problem, getting the best solution is much harder than approaching it.

Therefore, in cluster environment, we distribute exclusive mutual seeds to different compute node, to increase the possibility of getting best result. Take 100 compute nodes and 10000-verteices graph as an example, `GB_SeedDistribute()` will assign seeds with in range $[1, 100]$ for host 1, seeds within $[101, 200]$ for host 2, ... , seeds within $[901, 1000]$ for host 100.

Now we discuss how to run hundreds of independent instances in cluster environment. Refer to http://www.logos.ic.i.u-tokyo.ac.jp/phoenix/gxp_quick_man.shtml for GXP cluster shell's usage.

Since GXP provide `$GXP_HOST_IDX` and `$GXP_NUM_HOSTS` as environment variables, we use following shell scripts (`gxprun`) to passing them to our program.

```
2   #!/bin/bash
```

```
3   #Check arguments
4   hostid=‘expr $GXP_HOST_IDX + 1‘
5   hostnum=$GXP_NUM_HOSTS
6   prog=./main
7   file=$1
8   iter=$2
9   timeout=$3
10  threshold=$4
11  $prog $file $hostid $hostnum $iter $timeout $threshold
```

**Figure 12 GXP Scripts**

Starting up GXP on your clusters, type following command.

```
12    e {{./gxprun file iter timeout threshold }} cat > result
```

After result having been gathered to text file `result`, from which best result can be retrieved by using combination of "`grep`" and "`sort`".

### 3.2 Implementation Framework

The performance of graph bisection highly depends on the implementation, especially the choosing of data structure. The specific data structures and routines implemented APIs are in `include/gbimpl.h`.

#### 3.2.1 Data Structure and Global Variables

A universal data structure used by all routines is "bucket-list", similar as ones refered in [4], [6], see Figure 13. Insert and delete on this "bucket-list" structure are executed in $O(1)$ steps, only the update bucket bounds is $O(|E|)$. Since during bisection, the whole information of graph will be passed from one heuristic to another, making these information global readable is both efficient and simple. Other global variables include bisection information, timeout, threshold and other global flags to tuning heuristics.

#### 3.2.2 Routines

These global routines can be categorized as environment routines and bisection routines. Environment routines is in charge of initializing or finalizing the environment, such as memory allocation, variable initialization and bucket-list maintain, etc. Bisection routines are called for specific heuristic partitioning.

## 4 Experiments

Appendix gives the result of "real-world" benchmarks. Single Host 100-iterations shows the bisection result produced by Greedy Growing, Min-Max Greedy and Randomized Tabu Search, 100 iterations seperately. Table GXP Parallel RRTS gives a comparison of RRTS (100 iterations on single host, column RRTS-100) to

RRTS (100 iterations in 100 compute nodes, column P-RRTS-100), for time reason, we also set a timeout as 100 sec.

From the result, we can conclude that RRTS heuristc is good at finding the best result but time costing (Results with * means timeout exit, not iterations finished). And GXP Parallel RRTS significantly increase the performance with much better result. If given enought time, (P)-RRTS can approach the best best result with much higher possibility than other heuristics.

A complementary to RRTS time costing characteristic is multi-level graph partition[5], one of stream of current graph bisection. Multi-level graph partition is able to extremly reduce the cost time but reserve a good quality bisection.

# References


[1] T. N. Bui and C. Jones. Finding good approximate vertex and edge partitions is NP-hard. *Information Processing Letters*, 42(3):153-159, 1992.

[2] M.R. Garey and D.S. Johnson. Computers and Intractability, A guide to the theory of NP Completeness. *W.H. Freeman and Company*, New York, 1979.

[3] B. Kernighan and S. Lin. An Efficient Heuristic Procedure for Partitioning Graphs. *The Bell System Technical Journal*, 49(2):291-207, Feb. 1970.

[4] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. *In Proceedings of the nineteenth design automation conference*, 175-181, 1982.

[5] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359-392, 1998.

[6] R. Battiti and A. Bertossi, Greedy, Prohibition, and Reactive Heuristics for GraphPartitioning, *IEEE Transactions on Computers*, 48(4):361-385, 1999.

[7] G.R. Schreiber and O.C. Martin. Cut Size Statistics of Graph Bisection Heuristics, *manuscript in submission to SIAM J. Optimization*, 1997.

[8] R. Battiti, A. Bertossi and Cappelletti. Multilevel Reactive Tabu Search for Graph Partitioning. *Preprint UTM 554, Dip. Mat., Univ. Trento, Italy*, 1999.

[9] GXP: http://www.logos.ic.i.u-tokyo.ac.jp/phoenix/gxp_quick_man.shtml

[10] METIS: http://glaros.dtc.umn.edu/gkhome/views/metis/

[11] Inter Tools: http://rtm.science.unitn.it/intertools/graph-partitioning/

# Appendix: Real-world Benchmark

### Single Host 100-iterations for different heuristics

| Graph | $\|V\|$ | $\|E\|$ | best | GG | | MMG | | RTS | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | result | time | result | time | result | time |
| 3elt | 4720 | 13722 | 90 | 101 | 0.51 | 135 | 0.73 | 172 | 0.02 |
| 4elt | 15606 | 45878 | 140 | 151 | 1.96 | 214 | 3.21 | 253 | 0.08 |
| add20 | 2395 | 7462 | 609 | 760 | 0.27 | 798 | 0.50 | 938 | 0.02 |
| add32 | 4960 | 9462 | 11 | 11 | 0.53 | 39 | 0.71 | 55 | 0.03 |
| airfoil1 | 4253 | 12289 | 74 | 87 | 0.47 | 92 | 0.64 | 184 | 0.01 |
| bcspwr09 | 1723 | 2394 | 9 | 26 | 0.15 | 12 | 0.21 | 36 | 0.00 |
| bcsstk13 | 2003 | 40940 | 2355 | 2751 | 0.40 | 2477 | 0.78 | 2360 | 0.02 |
| bcsstk29 | 13992 | 302748 | 2843 | 5861 | 3.73 | 4938 | 6.53 | 6270 | 0.19 |
| bcsstk30 | 28924 | 1007284 | 6394 | 10430 | 10.30 | 6792 | 18.48 | 9347 | 0.59 |
| bcsstk31 | 35588 | 572914 | 3032 | 18596 | 8.00 | 3863 | 16.41 | 6662 | 0.53 |
| bcsstk32 | 44609 | 985046 | 5672 | 10446 | 11.84 | 10408 | 24.05 | 15592 | 0.81 |
| bcsstk33 | 8738 | 291583 | 10172 | 12066 | 3.07 | 10853 | 4.95 | 13010 | 0.17 |
| big | 15606 | 45878 | 142 | 151 | 1.99 | 201 | 3.15 | 188 | 0.08 |
| Breg100.20 | 100 | 150 | 15 | 18 | 0.00 | 16 | 0.01 | 18 | 0.00 |
| Breg100.4 | 100 | 150 | 4 | 4 | 0.00 | 4 | 0.00 | 8 | 0.00 |
| Breg100.8 | 100 | 150 | 8 | 8 | 0.00 | 8 | 0.01 | 16 | 0.00 |
| Breg5000.0 | 5000 | 7500 | 0 | 0 | 0.01 | 0 | 0.00 | 816 | 0.04 |
| Breg5000.16 | 5000 | 7500 | 16 | 32 | 0.52 | 16 | 1.36 | 802 | 0.03 |
| Breg5000.4 | 5000 | 7500 | 4 | 4 | 0.52 | 4 | 1.44 | 824 | 0.03 |
| Breg5000.8 | 5000 | 7500 | 7 | 10 | 0.52 | 8 | 1.44 | 814 | 0.04 |
| Breg500.0 | 500 | 750 | 0 | 0 | 0.00 | 0 | 0.00 | 6 | 0.00 |
| Breg500.12 | 500 | 750 | 12 | 18 | 0.04 | 16 | 0.05 | 12 | 0.00 |
| Breg500.16 | 500 | 750 | 15 | 20 | 0.04 | 18 | 0.05 | 42 | 0.00 |
| Breg500.20 | 500 | 750 | 20 | 40 | 0.04 | 32 | 0.05 | 68 | 0.00 |
| Cat.1052 | 1052 | 1051 | 1 | 1 | 0.08 | 1 | 0.12 | 3 | 0.00 |
| Cat.352 | 352 | 351 | 1 | 1 | 0.02 | 1 | 0.03 | 3 | 0.00 |
| Cat.5252 | 5252 | 5264 | 1 | 5 | 0.53 | 15 | 0.69 | 12 | 0.04 |
| Cat.702 | 702 | 701 | 1 | 1 | 0.05 | 1 | 0.07 | 3 | 0.00 |
| crack | 10240 | 30380 | 184 | 192 | 1.27 | 209 | 2.17 | 215 | 0.05 |
| cs4 | 22499 | 43858 | 397 | 533 | 3.08 | 442 | 9.51 | 560 | 0.25 |
| cti | 16840 | 48232 | 334 | 668 | 2.14 | 657 | 4.35 | 758 | 0.09 |
| data | 2851 | 15093 | 189 | 220 | 0.35 | 231 | 0.50 | 233 | 0.02 |
| DEBR12 | 4096 | 8189 | 548 | 564 | 0.42 | 684 | 0.94 | 700 | 0.03 |
| fe_4elt2 | 11143 | 32818 | 130 | 130 | 1.37 | 136 | 2.14 | 260 | 0.05 |
| fe_body | 45087 | 163734 | 304 | 766 | 6.80 | 512 | 16.48 | 1090 | 0.68 |
| fe_pwt | 36519 | 144794 | 364 | 362 | 5.25 | 403 | 8.40 | 1833 | 0.33 |
| fe_sphere | 16386 | 49152 | 388 | 386 | 2.13 | 422 | 3.65 | 480 | 0.10 |
| G1000.0025 | 1000 | 1272 | 95 | 126 | 0.08 | 130 | 0.16 | 128 | 0.00 |
| G1000.005 | 1000 | 2496 | 445 | 525 | 0.09 | 532 | 0.21 | 509 | 0.00 |
| G1000.01 | 1000 | 5064 | 1362 | 1496 | 0.12 | 1502 | 0.28 | 1433 | 0.00 |
| G1000.02 | 1000 | 10107 | 3382 | 3610 | 0.17 | 3616 | 0.38 | 3441 | 0.01 |

**Single Host 100-iterations for different heuristics (continue)**

| | | | | GG | | MMG | | RTS | |
|---|---|---|---|---|---|---|---|---|---|
| Graph | $\|V\|$ | $\|E\|$ | best | result | time | result | time | result | time |
| G124.02 | 124 | 149 | 13 | 14 | 0.01 | 14 | 0.01 | 19 | 0.00 |
| G124.04 | 124 | 318 | 63 | 69 | 0.01 | 66 | 0.01 | 67 | 0.00 |
| G124.08 | 124 | 620 | 178 | 189 | 0.01 | 188 | 0.01 | 188 | 0.00 |
| G124.16 | 124 | 1271 | 449 | 451 | 0.02 | 463 | 0.02 | 464 | 0.00 |
| G250.01 | 250 | 331 | 29 | 33 | 0.01 | 35 | 0.02 | 35 | 0.00 |
| G250.02 | 250 | 612 | 114 | 130 | 0.02 | 126 | 0.03 | 124 | 0.00 |
| G250.04 | 250 | 1283 | 357 | 385 | 0.02 | 386 | 0.04 | 381 | 0.00 |
| G250.08 | 250 | 2421 | 828 | 863 | 0.03 | 865 | 0.05 | 842 | 0.00 |
| G500.005 | 500 | 625 | 49 | 64 | 0.04 | 66 | 0.06 | 71 | 0.00 |
| G500.01 | 500 | 1223 | 218 | 253 | 0.04 | 262 | 0.07 | 252 | 0.00 |
| G500.02 | 500 | 2355 | 626 | 682 | 0.05 | 670 | 0.09 | 650 | 0.00 |
| G500.04 | 500 | 5120 | 1744 | 1837 | 0.08 | 1855 | 0.14 | 1784 | 0.00 |
| Grid1000.20 | 1000 | 1930 | 20 | 20 | 0.09 | 20 | 0.12 | 53 | 0.00 |
| Grid100.10 | 100 | 180 | 10 | 10 | 0.00 | 10 | 0.01 | 12 | 0.00 |
| Grid.4920 | 4920 | 9698 | 60 | 60 | 0.51 | 62 | 0.77 | 106 | 0.01 |
| Grid5000.50 | 5000 | 9850 | 50 | 50 | 0.52 | 50 | 0.80 | 155 | 0.01 |
| Grid500.21 | 500 | 955 | 21 | 21 | 0.04 | 21 | 0.05 | 25 | 0.00 |
| Grid.900 | 900 | 1740 | 30 | 30 | 0.08 | 30 | 0.11 | 36 | 0.00 |
| memplus | 17758 | 54196 | 6139 | 6557 | 2.34 | 7649 | 26.57 | 7439 | 0.70 |
| nasa4704 | 4707 | 50026 | 1292 | 1692 | 0.76 | 1352 | 1.29 | 1560 | 0.04 |
| RCat.134 | 134 | 133 | 1 | 1 | 0.00 | 1 | 0.01 | 1 | 0.00 |
| RCat.5114 | 5114 | 5118 | 1 | 3 | 0.49 | 3 | 0.66 | 3 | 0.06 |
| RCat.554 | 554 | 553 | 1 | 1 | 0.04 | 1 | 0.06 | 3 | 0.00 |
| RCat.994 | 994 | 993 | 1 | 1 | 0.07 | 1 | 0.11 | 1 | 0.00 |
| U1000.05 | 1000 | 2394 | 1 | 7 | 0.10 | 1 | 0.13 | 13 | 0.00 |
| U1000.10 | 1000 | 4696 | 39 | 82 | 0.12 | 48 | 0.17 | 64 | 0.00 |
| U1000.20 | 1000 | 2393 | 222 | 306 | 0.16 | 273 | 0.23 | 257 | 0.01 |
| U1000.40 | 1000 | 18015 | 737 | 873 | 0.24 | 754 | 0.34 | 1129 | 0.01 |
| U500.05 | 500 | 1282 | 2 | 6 | 0.04 | 2 | 0.06 | 13 | 0.00 |
| U500.10 | 500 | 2355 | 26 | 77 | 0.05 | 31 | 0.08 | 56 | 0.00 |
| U500.20 | 500 | 4549 | 178 | 180 | 0.07 | 184 | 0.11 | 184 | 0.00 |
| U500.40 | 500 | 8793 | 409 | 418 | 0.11 | 412 | 0.16 | 412 | 0.00 |
| uk | 4824 | 6837 | 23 | 36 | 0.48 | 33 | 0.67 | 46 | 0.02 |
| vibrobox | 12328 | 165250 | 10343 | 15858 | 2.64 | 11785 | 7.74 | 12508 | 0.18 |
| W-grid1000.40 | 1000 | 2000 | 40 | 40 | 0.09 | 40 | 0.12 | 40 | 0.00 |
| W-grid100.20 | 100 | 200 | 20 | 30 | 0.00 | 20 | 0.01 | 26 | 0.00 |
| W-grid5000.100 | 5000 | 1 | 100 | 100 | 0.53 | 100 | 0.80 | 138 | 0.02 |
| W-grid500.42 | 500 | 1000 | 40 | 64 | 0.04 | 42 | 0.06 | 46 | 0.00 |
| whitaker3 | 9800 | 28989 | 128 | 128 | 1.19 | 131 | 1.77 | 157 | 0.04 |
| wing | 62032 | 121544 | 950 | 1313 | 10.56 | 1230 | 58.20 | 1776 | 2.42 |
| wing_nodal | 10937 | 75488 | 1708 | 1803 | 1.91 | 1852 | 3.57 | 1947 | 0.11 |

## GXP Parallel RRTS

| Graph | $\|V\|$ | $\|E\|$ | best | RRTS-100 | | P-RRTS-100 | |
|---|---|---|---|---|---|---|---|
| | | | | result | time | result | time |
| 3elt | 4720 | 13722 | 90 | 90 | 56.21 | 90 | 74.87 |
| 4elt | 15606 | 45878 | 140 | 100 | 174.49 | 140 | 100.64 |
| add20 | 2395 | 7462 | 609 | 715 | 30.71 | 609 | 41.78 |
| add32 | 4960 | 9462 | 11 | 20 | 47.99 | 11 | 62.49 |
| airfoil1 | 4253 | 12289 | 74 | 77 | 50.49 | 74 | 66.433 |
| bcspwr09 | 1723 | 2394 | 9 | 9 | 13.67 | 9 | 17.53 |
| bcsstk13 | 2003 | 40940 | 2355 | 2355 | 100.00 | 2355 | 100.651 |
| bcsstk29 | 13992 | 302748 | 2843 | 5927* | 100.03 | 3047* | 102.84 |
| bcsstk30 | 28924 | 1007284 | 6394 | 10051* | 100.05 | 6394* | 108.224 |
| bcsstk31 | 35588 | 572914 | 3032 | 3828* | 100.07 | 3781* | 100.64 |
| bcsstk32 | 44609 | 985046 | 5672 | 6379* | 100.05 | 6096* | 116.01 |
| bcsstk33 | 8738 | 291583 | 10171 | 10171* | 100.01 | 10171* | 102.22 |
| big | 15606 | 45878 | 140 | 187* | 100.00 | 140* | 101.126 |
| Breg100.20 | 100 | 150 | 15 | 16 | 0.68 | 16 | 1.01 |
| Breg100.4 | 100 | 150 | 4 | 4 | 0.67 | 4 | 1.06 |
| Breg100.8 | 100 | 150 | 8 | 8 | 0.67 | 8 | 0.97 |
| Breg5000.0 | 5000 | 7500 | 0 | 0 | 3.15 | 0 | 16.79 |
| Breg5000.16 | 5000 | 7500 | 16 | 16 | 63.66 | 16 | 99.142 |
| Breg5000.4 | 5000 | 7500 | 4 | 4 | 49.95 | 4 | 100.16 |
| Breg5000.8 | 5000 | 7500 | 7 | 8 | 46.3 | 8 | 74.35 |
| Breg500.0 | 500 | 750 | 0 | 0 | 0.61 | 0 | 1.05 |
| Breg500.12 | 500 | 750 | 12 | 12 | 3.75 | 12 | 4.99 |
| Breg500.16 | 500 | 750 | 15 | 16 | 3.77 | 16 | 5.01 |
| Breg500.20 | 500 | 750 | 20 | 20 | 3.78 | 20 | 4.98 |
| Cat.1052 | 1052 | 1051 | 1 | 1 | 6.86 | 1 | 8.950 |
| Cat.352 | 352 | 351 | 1 | 1 | 2.15 | 1 | |
| Cat.5252 | 5252 | 5264 | 1 | 5 | 39.61 | 5 | 55.81 |
| Cat.702 | 702 | 701 | 1 | 1 | 4.46 | 1 | 5.88 |
| crack | 10240 | 30380 | 184 | 185* | 100.00 | 184* | 101.15 |
| cs4 | 22499 | 43858 | 397 | 519* | 100.00 | 397* | 101.91 |
| cti | 16840 | 48232 | 334 | 365* | 100.00 | 334* | 101.31 |
| data | 2851 | 15093 | 189 | 192 | 48.93 | 189 | 67.84 |
| DEBR12 | 4096 | 8189 | 548 | 570 | 44.15 | 562 | 61.37 |
| fe_4elt2 | 11143 | 32818 | 130 | 120* | 100.00 | 130* | 101.32 |
| fe_body | 45087 | 163734 | 304 | 953* | 100.00 | 319* | 102.66 |
| fe_pwt | 36519 | 144794 | 341 | 360* | 100.01 | 341* | 102.46 |
| fe_sphere | 16386 | 49152 | 386 | 396* | 100.00 | 386* | 102.31 |
| G1000.0025 | 1000 | 1272 | 95 | 107 | 7.66 | 102 | 9.948 |
| G1000.005 | 1000 | 2496 | 445 | 462 | 11.22 | 456 | 15.468 |
| G1000.01 | 1000 | 5064 | 1362 | 1391 | 18.54 | 1371 | 25.49 |
| G1000.02 | 1000 | 10107 | 3382 | 3408 | 34.17 | 3391 | 45.11 |

| Graph | $\|V\|$ | $\|E\|$ | best | RRTS-100 | | P-RRTS-100 | |
|---|---|---|---|---|---|---|---|
| | | | | result | time | result | time |
| G124.02 | 124 | 149 | 13 | 13 | 0.83 | 13 | 1.213 |
| G124.04 | 124 | 318 | 63 | 63 | 1.24 | 63 | 2.075 |
| G124.08 | 124 | 620 | 178 | 178 | 2.02 | 178 | 2.91 |
| G124.16 | 124 | 1271 | 449 | 449 | 3.74 | 449 | 6.77 |
| G250.01 | 250 | 331 | 29 | 29 | 1.77 | 29 | 2.62 |
| G250.02 | 250 | 612 | 114 | 114 | 2.52 | 114 | 3.60 |
| G250.04 | 250 | 1283 | 357 | 357 | 4.30 | 357 | 5.975 |
| G250.08 | 250 | 2421 | 828 | 828 | 7.41 | 828 | 10.024 |
| G500.005 | 500 | 625 | 49 | 56 | 3.64 | 52 | 4.85 |
| G500.01 | 500 | 1223 | 218 | 220 | 4.92 | 219 | 6.95 |
| G500.02 | 500 | 2355 | 626 | 650 | 7.91 | 626 | 11.73 |
| G500.04 | 500 | 5120 | 1744 | 1751 | 15.41 | 1744 | 21.64 |
| Grid1000.20 | 1000 | 1930 | 20 | 20 | 8.07 | 20 | 11.62 |
| Grid100.10 | 100 | 180 | 10 | 10 | 0.69 | 10 | 1.14 |
| Grid.4920 | 4920 | 9698 | 60 | 60 | 49.03 | 60 | 77.94 |
| Grid5000.50 | 5000 | 9850 | 50 | 50 | 49.65 | 50 | 80.60 |
| Grid500.21 | 500 | 955 | 21 | 21 | 3.87 | 21 | 5.72 |
| Grid.900 | 900 | 1740 | 30 | 30 | 7.3 | 30 | 10.50 |
| memplus | 17758 | 54196 | 6139 | 7297* | 100.00 | 6139* | 102.18 |
| nasa4704 | 4707 | 50026 | 1292 | 1292 | 100.00 | 1292 | 101.403 |
| RCat.134 | 134 | 133 | 1 | 1 | 0.72 | 1 | 1.22 |
| RCat.5114 | 5114 | 5118 | 1 | 3 | 33.8 | 3 | 90.63 |
| RCat.554 | 554 | 553 | 1 | 1 | 3.17 | 1 | 9.98 |
| RCat.994 | 994 | 993 | 1 | 1 | 13.78 | 1 | 18.65 |
| U1000.05 | 1000 | 2394 | 1 | 1 | 9.93 | 1 | 13.92 |
| U1000.10 | 1000 | 4696 | 39 | 39 | 15.66 | 39 | 22.44 |
| U1000.20 | 1000 | 2393 | 222 | 222 | 27.18 | 222 | 39.232 |
| U1000.40 | 1000 | 18015 | 737 | 737 | 49.60 | 737 | 72.419 |
| U500.05 | 500 | 1282 | 2 | 2 | 4.96 | 2 | 7.176 |
| U500.10 | 500 | 2355 | 26 | 26 | 7.62 | 26 | 11.01 |
| U500.20 | 500 | 4549 | 178 | 178 | 12.97 | 178 | 18.77 |
| U500.40 | 500 | 8793 | 409 | 412 | 22.88 | 412 | 33.96 |
| uk | 4824 | 6837 | 23 | 30 | 35.89 | 24 | 51.01 |
| vibrobox | 12328 | 165250 | 10343 | 11868* | 100.02 | 10343* | 102.43 |
| W-grid1000.40 | 1000 | 2000 | 40 | 40 | 8.32 | 40 | 15.29 |
| W-grid100.20 | 100 | 200 | 20 | 20 | 0.72 | 20 | 1.78 |
| W-grid5000.100 | 5000 | 10000 | 100 | 100 | 59.75 | 100 | 84.23 |
| W-grid500.42 | 500 | 1000 | 40 | 42 | 4.07 | 40 | 11.84 |
| whitaker3 | 9800 | 28989 | 128 | 128* | 100.00 | 127* | 101.23 |
| wing | 62032 | 121544 | 950 | 1776* | 2.42 | 1007* | 104.29 |
| wing_nodal | 10937 | 75488 | 1708 | 1711* | 100.45 | 1708* | 101.82 |