Science & Technology Facilities Council

Technical Report
RAL-TR-2016-004

Nested dissection revisited

C Ashcraft, I Duff, J Hogg, JA Scott, S Thorne

April 2016

Enquiries concerning this report should be addressed to:

RAL Library
STFC Rutherford Appleton Laboratory
Harwell Oxford
Didcot
OX11 0QX

Tel: +44(0)1235 445384
Fax: +44(0)1235 446403
email: libraryral@stfc.ac.uk

Science and Technology Facilities Council reports are available online at: http://epubs.stfc.ac.uk

ISSN 1358-6254

Nested dissection revisited

Cleve Ashcraft[1], Iain Duff[2], Jonathan Hogg[2], Jennifer Scott[2] and Sue Thorne[2]

ABSTRACT

Nested dissection algorithms are widely used to order large sparse matrices with a symmetric sparsity pattern prior to using a sparse direct linear solver. In this report, we revisit nested dissection algorithms. In particular, level based methods and multilevel methods are examined, along with techniques for refining and improving the ordering. Our experience has led to the development of a new open source nested dissection-based ordering package `SPRAL_ND`. We report on the design of `SPRAL_ND`, the options that it offers and illustrate its performance using problems arising from a range of practical applications. Comparisons are made with the widely used METIS package. We find that while multilevel methods generally produce the best orderings in terms of the subsequent fill in the matrix factor and the flops needed to compute it, using a non-multilevel ordering that is significantly less expensive to compute can result in a reduction in the total solution time when used with a state-of-the-art parallel sparse direct solver.

Keywords: nested dissection, recursive bisection, sparse symmetric matrices, sparse matrix ordering.

AMS(MOS) subject classifications: 65F30, 65F50

---

[1] Livermore Software Technology Corporation, 7374 Las Positas Road, Livermore, CA 94550, USA.

[2] Scientific Computing Department, STFC Rutherford Appleton Laboratory, Harwell Campus, Oxfordshire, OX11 0QX, UK.

Correspondence to: jennifer.scott@stfc.ac.uk

The work of Duff, Hogg, Scott and Thorne was supported by EPSRC grants EP/I013067/1 and EP/M025179/1.

April 4, 2016

1 Background and motivation

When solving a sparse linear system of equations $Ax = b$ using direct methods, the efficiency of the direct factorization of A both in terms of operations and storage for the factors, is very dependent on the ordering of the rows and columns used when performing the factorization. In particular, reordering the rows and columns of a sparse symmetric positive-definite matrix A prior to computing its Cholesky factorization LL^T can significantly reduce the number of nonzeros in L and hence the storage for L, the work required for the factorization, and the work in solving the triangular systems $Ly = b$ and $L^T x = y$ that complete the solution process. Finding the optimal ordering is an NP-complete problem [37] so, instead of looking for this, we use heuristics to find a "good" ordering.

Since the original work of Markowitz [30] in 1957, this has been the subject of much research. An important class of ordering methods is based upon the minimum degree algorithm, which was first proposed as algorithm S2 by Tinney and Walker [36]. This uses a local strategy: at each stage of the elimination process, the diagonal entry in the row of the remaining submatrix with the least number of entries is chosen as the next pivot. Variants include Liu's multiple minimum degree algorithm [29] and the approximate minimum degree (AMD) algorithm of Amestoy, Davis and Duff [2] (see also [13]).

An alternative to a local strategy is to compute an ordering using a global strategy. Methods based on nested dissection are particularly popular. Nested dissection was first introduced by George [16] in the early 1970s and had its roots in finite-element substructuring. The central concept in a nested dissection ordering is the removal of a set of vertices (called a *separator*) from a graph $\mathcal{G}$ associated with the matrix A that leaves the remaining graph in two (or more) disconnected parts. These parts are themselves further divided by the removal of sets of vertices, with the dissection nested to any depth. The quality of a nested dissection ordering depends crucially upon the quality of its separators. In the mid-1990's, a number of efficient implementations of nested dissection ordering algorithms were designed and developed, most notably CHACO from Hendrickson and Rothberg [20], MeTiS from Karypis and Kumar [25, 27], and SCOTCH from Pellegrini [32] (although we note that all these codes are primarily for graph partitioning). It has been shown experimentally that, when used with a modern sparse direct solver, nested dissection orderings can be significantly more effective than minimum-degree methods, particularly for very large problems and problems arising from three-dimensional finite-element applications (see, for example, the experiments reported by Duff and Scott [14]). They also generally provide a more balanced elimination tree, and this is beneficial for parallel direct solvers.

Many sparse matrices can be reordered using level set based algorithms such as the well-known Cuthill-McKee algorithm [10] to have a form in which all nonzeros are contained within a relatively narrow band. It is then straightforward to find a good separator and indeed subsequent good separators can be found in a nested fashion to generate a nested dissection ordering of the original matrix.

The original approach to nested dissection [16] and the algorithms described in the book by George and Liu [17] employed level set based methods. However, most current partitioners and algorithms for nested dissection use multilevel techniques. One reason that level set based methods fell out of favour and multilevel techniques have been ubiquitous for the last twenty years is that the former can struggle to perform well on irregularly structured matrices that cannot be permuted to have a narrow band. Moreover, while more sophisticated partitioning techniques like spectral bisection [6] have been used to obtain high quality separators, they are expensive to run on the original matrix rather than on a much smaller matrix obtained through the use of multilevel reduction. A key objective of our work is to revisit both level set based methods and multilevel methods. By doing this we hope to be able to exploit the matrix structure to obtain a less costly dissection when the structure allows this.

Our study has led to the design and development of a new nested dissection package `SPRAL_ND`. An important motivation behind this study and the development of `SPRAL_ND` is that we want to incorporate the code as an ordering into our sparse direct solvers avoiding any licensing issues and the added complexity of a general partitioning code. Our aim was to develop a code that offers an open source alternative to MeTiS that is flexible in the options that it offers and is efficient and able to compute good quality

orderings. In the future, we intend to use our code as a research vehicle to test different approaches and to develop versions that run well on parallel architectures.

The remainder of this report is organised as follows. We define our terminology in Section 2, introducing the graph theory terms that we will use later in the paper and discussing the partitioning problem from both a matrix and a graph perspective. We describe the general nested dissection framework in Section 3 and the multilevel approach in Section 5. We consider finding an initial separator in Section 4 using either a level-set approach or a novel half-level approach. In Section 6, we consider various methods for refining and improving a partition and then summarise our nested dissection algorithm implemented within `SPRAL_ND` in Section 7. In Section 8, we present some numerical experiments on an extensive set of test problems from the University of Florida Sparse Matrix Collection [12] and compare our implementation with using MeTiS for obtaining a nested dissection ordering for use with a sparse direct solver. Finally, some concluding remarks are made in Section 9.

2 Definitions and terminology

In this section, we introduce the notation that we will use throughout this report. Note that we only use the pattern of the matrix and not its numerical values. It is convenient when describing partitionings and nested dissection orderings to use the equivalence between sparse matrices and graphs. If the sparse symmetric matrix $A = \{a_{ij}\}$ is of order n, the *adjacency graph* $\mathcal{G} = \mathcal{G}(\mathcal{V}, \mathcal{E})$ of A is an undirected graph with *vertices* (or *nodes*) $\mathcal{V} = \{1, \ldots, n\}$ and edges $\mathcal{E}$, where an *edge* (u, v) is present in $\mathcal{E}$ if and only if $a_{uv} \neq 0$ $(u \neq v)$. Vertices u and v are *adjacent* vertices (or *neighbours*) in $\mathcal{G}$ if the edge (u, v) is present in $\mathcal{E}$. A *path* of length k in $\mathcal{G}$ is an ordered set of distinct vertices $\{v_1, v_2, \ldots, v_k, v_{k+1}\}$ subject to $(v_j, v_{j+1}) \in \mathcal{E}$ for $j = 1, \ldots, k$. If there is a path between all pairs of vertices in $\mathcal{G}$, the graph is *connected* and the matrix A is *irreducible* (or *non-separable*). We will assume throughout our discussions that $\mathcal{G}$ is connected; if not, `SPRAL_ND` applies our algorithms to each component independently.

We will often associate weights to vertices and less frequently to edges. We denote by $|v|$ the weight of vertex v. The weight of a subset of vertices $\mathcal{U} = \{u_1, \ldots, u_k\}$ is the sum of the weights of the vertices in the subset,

$$|\mathcal{U}| = \sum_{i=1}^{k} |u_i|.$$

The goal of nested dissection is to partition $\mathcal{G}(\mathcal{V}, \mathcal{E})$; the partitioning ϕ is then used to order the underlying sparse matrix. The permutation vector corresponding to ϕ will be denoted by π. Two commonly used ways of obtaining a partition are to use a *vertex separator* or an *edge separator*.

- A vertex separator $\mathcal{S}$ partitions $\mathcal{V}$ so that

$$\mathcal{V} = \mathcal{S} \sqcup \mathcal{B}_1 \sqcup \ldots \sqcup \mathcal{B}_k. \tag{2.1}$$

  Note that here and elsewhere, we use the notation $\sqcup$ to denote a disjoint union (that is $\mathcal{B}_i \cap \mathcal{B}_j = 0, i \neq j$ and $\mathcal{B}_i \cap \mathcal{S} = 0$, for all i). When $k = 2$, $\mathcal{S}$ is a bisector and this is the only case we consider. By removing the rows/columns in $\mathcal{S}$ from A, a reducible matrix is obtained.

- An edge separator $\mathcal{E}_\mathcal{S}$ produces a partition of $\mathcal{V}$

$$\mathcal{V} = \mathcal{B}_1 \sqcup \ldots \sqcup \mathcal{B}_k,$$

  where

$$\mathcal{E}_\mathcal{S} = \{(u, v) \mid u \in \mathcal{B}_i, v \in \mathcal{B}_j, i \neq j\}.$$

Given an edge separator, it is trivial to construct a vertex separator although there is considerable freedom in the choice of the vertex separator. The well-known MeTiS package [25, 27] proceeds by first

constructing an edge separator. However, based on some early experiments, we have chosen to focus on constructing a vertex separator directly; this is also the approach used in the BEND package [20] and the recent MORSy software [33]. Throughout the remainder of this paper, the term *separator* will refer to a vertex separator. Furthermore, for convenience of notation, in the case of bisection ($k = 2$), we will denote $\mathcal{B}_1$ by $\mathcal{B}$ and $\mathcal{B}_2$ by $\mathcal{W}$. These sets are commonly called "black" and "white", respectively.

A separator is *minimal* if the removal of any vertex from the set causes the resulting set to not be a separator. We use the term *wide separator* for a separator that is not minimal.

Given $(\mathcal{B}, \mathcal{W}, \mathcal{S})$, we define the partition vector $\phi(\mathcal{B}, \mathcal{W}, \mathcal{S})$ to have entries

$$\phi(\mathcal{B}, \mathcal{W}, \mathcal{S})(i) = \begin{cases} 0, & \text{if } i \in \mathcal{S}, \\ 1, & \text{if } i \in \mathcal{B}, \\ 2, & \text{if } i \in \mathcal{W}. \end{cases}$$

We observe that the values $0, 1, 2$ are not significant: we just need a different value for vertices belonging to each of the three sets. For simplicity of notation, we will often denote $\phi(\mathcal{B}, \mathcal{W}, \mathcal{S})$ by ϕ.

Once we have computed a partition, we have to decide whether or not it is acceptable. Consider partitioning $A = LL^T$ into the form

$$\begin{bmatrix} A_{\mathcal{B},\mathcal{B}} & & A_{\mathcal{B},\mathcal{S}} \\ & A_{\mathcal{W},\mathcal{W}} & A_{\mathcal{W},\mathcal{S}} \\ A_{\mathcal{B},\mathcal{S}}^T & A_{\mathcal{W},\mathcal{S}}^T & A_{\mathcal{S},\mathcal{S}} \end{bmatrix} = \begin{bmatrix} L_{\mathcal{B},\mathcal{B}} & & \\ & L_{\mathcal{W},\mathcal{W}} & \\ L_{\mathcal{B},\mathcal{S}} & L_{\mathcal{W},\mathcal{S}} & L_{\mathcal{S},\mathcal{S}} \end{bmatrix} \begin{bmatrix} L_{\mathcal{B},\mathcal{B}}^T & & L_{\mathcal{B},\mathcal{S}}^T \\ & L_{\mathcal{W},\mathcal{W}}^T & L_{\mathcal{W},\mathcal{S}}^T \\ & & L_{\mathcal{S},\mathcal{S}}^T \end{bmatrix} \tag{2.2}$$

where $L_{\mathcal{S},\mathcal{S}}$ is the Cholesky factor of the Schur complement

$$Z_{\mathcal{S},\mathcal{S}} = A_{\mathcal{S},\mathcal{S}} - A_{\mathcal{B},\mathcal{S}}^T A_{\mathcal{B},\mathcal{B}}^{-1} A_{\mathcal{B},\mathcal{S}} - A_{\mathcal{W},\mathcal{S}}^T A_{\mathcal{W},\mathcal{W}}^{-1} A_{\mathcal{W},\mathcal{S}}.$$

If either $A_{\mathcal{B},\mathcal{B}}$ or $A_{\mathcal{W},\mathcal{W}}$ (or both) is irreducible and all columns of $A_{\mathcal{B},\mathcal{S}}$ and $A_{\mathcal{W},\mathcal{S}}$ contain at least one nonzero entry, $Z_{\mathcal{S},\mathcal{S}}$ is dense and $L_{\mathcal{S},\mathcal{S}}$ is dense below its diagonal. Hence, it is often desirable to choose the partition to minimise $|\mathcal{S}|$.

The independence of the submatrices $A_{\mathcal{B},\mathcal{B}}$ and $A_{\mathcal{W},\mathcal{W}}$ can be exploited when factorizing A in parallel. Once $L_{\mathcal{B},\mathcal{B}}$ and $L_{\mathcal{W},\mathcal{W}}$ have been computed, $Z_{\mathcal{S},\mathcal{S}}$ must be computed and factorized. The amount of communication needed to form $Z_{\mathcal{S},\mathcal{S}}$ is related to $|\mathcal{S}|$. If A is factorized in parallel, we want to try to balance the load on each processor. This gives us another possible aim when forming our partition ϕ: we might want to choose $\mathcal{B}$, $\mathcal{W}$ and $\mathcal{S}$ such that $|\mathcal{B}| \approx |\mathcal{W}|$. Clearly, trying to balance $|\mathcal{B}|$ and $|\mathcal{W}|$ may be at odds with the aim of minimising $|\mathcal{S}|$ and may not balance the work involved in factorizing $A_{\mathcal{B},\mathcal{B}}$ and $A_{\mathcal{W},\mathcal{W}}$.

We can control the balance of the partition by using an upper bound on the imbalance imb. If we define

$$\mathsf{imb} = \frac{\max\left(|\mathcal{B}|, |\mathcal{W}|\right)}{\min\left(|\mathcal{B}|, |\mathcal{W}|\right)}, \tag{2.3}$$

a partition ϕ is said to be *acceptable* if $\mathsf{imb} \leq \alpha$ for a chosen balance parameter α. We have found it can be advantageous to allow significant imbalance; in `SPRAL_ND` the default setting for the balance parameter α is 4.0 (see Section 8.4). When choosing between partitions, one possibility (see [20]) is to choose an acceptable partition that minimizes

$$\frac{|\mathcal{S}|}{|\mathcal{B}|\,|\mathcal{W}|}.$$

Hence, we introduce the following evaluation function $\mathsf{cost1}$ that we want to minimise:

$$\mathsf{cost1}(\mathcal{B}, \mathcal{W}, \mathcal{S}) = \begin{cases} \dfrac{|\mathcal{S}|}{|\mathcal{B}|\,|\mathcal{W}|}, & \text{if } \mathsf{imb} \leq \alpha \\ \\ |\mathcal{V}| - 2 + \frac{|\mathcal{S}|}{|\mathcal{B}|\,|\mathcal{W}|}, & \text{otherwise.} \end{cases} \tag{2.4}$$

Note that

$$|\mathcal{V}| - 2 \geq \frac{|\mathcal{S}|}{|\mathcal{B}|\,|\mathcal{W}|},$$

for all possible $\mathcal{B}$, $\mathcal{W}$ and $\mathcal{S}$, and so an acceptable partition will always have a lower cost than an unacceptable one. If we have two partitions ϕ_1 and ϕ_2 and both are acceptable, or neither is acceptable, we choose the ϕ_i for which `cost1` is smaller. Otherwise, we choose the acceptable partition.

Many other evaluation functions have been used in the literature see, for example, [4,31]. In our study, we consider a second function for evaluating a partition

$$\mathsf{cost2}(\mathcal{B},\mathcal{W},\mathcal{S}) = \begin{cases} |\mathcal{S}|\,(1+\beta\,|\mathsf{diff}|)\,, & \text{if } \mathsf{imb} \leq \alpha \\ \\ |\mathcal{V}|\,(1+\beta) + |\mathcal{S}|\,(1+\beta\,|\mathsf{diff}|)\,, & \text{otherwise,} \end{cases} \tag{2.5}$$

where $\beta \geq 0$ is an imbalance penalty and

$$\mathsf{diff} = \frac{|\mathcal{B}| - |\mathcal{W}|}{|\mathcal{V}|}.$$

When $\beta = 0$, imbalance between $\mathcal{B}$ and $\mathcal{W}$ is ignored; when $\beta > 0$, imbalance increases the cost of the partition. Similarly to `cost1`, note that

$$|\mathcal{V}|\,(1+\beta) \geq |\mathcal{S}|\,(1+\beta\,|\mathsf{diff}|)\,,$$

for all possible $\mathcal{B}$, $\mathcal{W}$ and $\mathcal{S}$, and, hence, an acceptable partition will always have a lower cost than an unacceptable one.

3 The nested dissection algorithm

As already discussed, given a symmetric matrix A with adjacency graph $\mathcal{G}(\mathcal{V},\mathcal{E})$, the basic idea of nested dissection is to find a separator $\mathcal{S}$ and subsets $\mathcal{B}$ and $\mathcal{W}$ of $\mathcal{V}$ such that (2.1) holds with $k = 2$ and $\mathcal{B} = \mathcal{B}_1$ and $\mathcal{W} = \mathcal{B}_2$. If the rows and columns corresponding to the entries of $\mathcal{S}$ are ordered after those corresponding to $\mathcal{B}$ and $\mathcal{W}$, no fill occurs in the Cholesky factor of A in the off-diagonal blocks of the submatrix corresponding to $\mathcal{B}$ and $\mathcal{W}$ (see (2.2)). $\mathcal{B}$ and $\mathcal{W}$ can be reordered by applying the dissection strategy recursively. In practice, this dissection is applied recursively until the limit on the number of recursions has been reached (the default setting in `SPRAL_ND` is 20), the size of the submatrix is less than some prescribed threshold (the default setting in `SPRAL_ND` is 50), or the partitioning algorithm fails to find a valid partition (e.g. the graph has become fully connected). A variant of approximate minimum degree (denoted $\mathtt{AMD}(\mathcal{V},\mathcal{E})$) is then employed to order any remaining parts. In practice, we rarely reach the limit on the number of recursions. The general form of the nested dissection algorithm is summarised in Algorithm 1: it returns a permutation vector π. The parameter *PartitionAlg* specifies the algorithm to be used in determining the partition $\phi(\mathcal{B},\mathcal{W},\mathcal{S})$. It may be either `LevelSetAlg`, `HalfLevelAlg` or `MultilevelAlg` (that uses `LevelSetAlg` or `HalfLevelAlg` internally). These constituent algorithms will be described as Algorithms 3, 4 and 5.

3.1 The removal of dense rows

It is well known that sparse matrix ordering algorithms can be inefficient if the matrix has some rows and columns that are dense (or almost) dense. A number of approaches have been proposed to try and circumvent this problem for AMD orderings [1,8,11,13]. We follow the approach of Dollar and Scott [13] and include a preprocessing step that identifies rows and columns that are classified as dense and removes them before the ordering algorithm is applied to the remaining rows and columns. The vertices corresponding to the dense rows and columns are appended to the ordering for the reduced matrix. The criteria used in `SPRAL_ND` to classify rows and columns as dense is as in the sparse ordering package `HSL_MC68` from the HSL mathematical software library [23].

Algorithm 1 Nested dissection algorithm

`function` $\pi =$ `nested_dissection`$(A, PartitionAlg)$

Let A have adjacency graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$

if dissection has terminated **then**

$\pi =$ `AMD`$(\mathcal{V}, \mathcal{E})$

else

$\phi(\mathcal{B}, \mathcal{W}, \mathcal{S}) = PartitionAlg(\mathcal{V}, \mathcal{E})$

$\pi_{\mathcal{B}} =$ `nested_dissection`$(A_{\mathcal{B},\mathcal{B}},$ `PartitionAlg`$)$

$\pi_{\mathcal{W}} =$ `nested_dissection`$(A_{\mathcal{W},\mathcal{W}},$ `PartitionAlg`$)$

$\pi_{\mathcal{S}}$ is an ordering of $\mathcal{S}$

$$\pi = \begin{bmatrix} \pi_{\mathcal{B}} \\ \pi_{\mathcal{W}} \\ \pi_{\mathcal{S}} \end{bmatrix}$$

end if

3.2 The compressed graph

Many graphs that arise from practical applications (including finite-element analysis) contain vertices with the same adjacency structures (defined to be the set of vertices comprising the vertex together with its neighbours). If these vertices can be identified, a compressed graph can be constructed that provides a more concise graph representation of the original matrix. The compressed graph is formed by merging all the vertices with the same adjacency structures into a single weighted vertex (this is often termed a supervariable). We set the vertex weight of the new vertex to be equal to the number of vertices that are merged into it and each edge (u, v) has edge weight equal to the product of the weights of the endpoint vertices u and v. The advantage of using a compressed graph is that it can have significantly fewer vertices and edges, thereby reducing the memory and time required to compute a nested dissection ordering [3,34].

A number of algorithms for identifying supervariables have been proposed. Several of the sparse matrix packages within the HSL library employ different implementations (see [22] for details). In our nested dissection code, we follow the approach used by the package `HSL_MC78`. Note that, although our implementation is designed to be efficient, its use incurs an overhead so that, if the user knows that his or her problem has no vertices with identical adjacency structures, the option within `SPRAL_ND` to compress the graph should not be selected.

4 Finding an initial separator

In this section, we discuss two ways of computing an initial separator for an adjacency graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ using level set techniques. For this we need some further definitions. The *distance* $\mathsf{dist}(u, v)$ between two vertices u and v is the length of the shortest path between them. The *diameter* is the maximum distance between any two vertices.

$$\mathsf{diam}(\mathcal{G}) = \max_{u,v \in \mathcal{V}} \mathsf{dist}(u, v).$$

A vertex v is *extremal* with respect to a vertex u if v is as far away from u as possible

$$\mathsf{dist}(u, v) = \max_{w} \mathsf{dist}(u, w).$$

When two vertices u and v are *mutually extremal*, then they form a *pseudo-diameter* pair

$$\mathsf{dist}(u,v) = \max_w \mathsf{dist}(u,w) = \max_w \mathsf{dist}(v,w)$$

and u and v are both *pseudo-peripheral* vertices.

Given any vertex $s \in \mathcal{V}$, the vertices can be partitioned into *level sets*

$$\mathcal{V} = \mathcal{L}_0(s) \sqcup \mathcal{L}_1(s) \sqcup \ldots \sqcup \mathcal{L}_k(s)$$

where $v \in \mathcal{L}_i(s)$ if and only if $\mathsf{dist}(s,v) = i$. Vertex s is referred to as the *root* vertex and

$$\mathcal{L}(s) = \{\mathcal{L}_0(s), \mathcal{L}_1(s), \ldots, \mathcal{L}_k(s)\}$$

is a *rooted level-set structure* of depth k.

It is easy to see that each level set $\mathcal{L}_i(s)$, $i = 1, \ldots, k-1$, defines a vertex separator. An edge (u,v) in the graph occurs only when u and v belong to the same level set or to adjacent level sets.

Algorithm 2 describes a simple algorithm to find a pseudo-diameter pair of vertices. A more robust and sophisticated method used by Reid and Scott [34] is a modification of the Gibbs-Poole-Stockmeyer algorithm [18]. Using a pseudo-peripheral vertex s as the root node generally leads to a long thin level-set structure $\mathcal{L}(s)$.

Algorithm 2 Algorithm to find a pseudo-diameter pair (s,t)

```
function (s,t) = PseudoPairAlg(V,E)

choose root s ∈ V at random
construct L(s) with depth k_s
loop
  find t ∈ V that is extremal w.r.t. s
  construct L(t) with depth k_t ≥ k_s
  if k_s = k_t then
    return
  end if
  replace root vertex s ← t
end loop
```

4.1 A single root level set approach

In their book [17], George and Liu provide an implementation `GENND` of nested dissection that uses a level-set approach. Having constructed $\mathcal{L}(s)$, they take as the separator $\mathcal{S}$ the level set $\mathcal{L}_j(s)$, where $j = \lfloor (k+1)/2 \rfloor$, and then move any vertices that are not adjacent to any vertices in $\mathcal{L}_{j+1}(s)$ since their removal still leaves a separator.

We strengthen their approach by examining each of the $k-1$ partitions $\phi_j = \phi(\mathcal{B}_j, \mathcal{W}_j, \mathcal{S}_j)$ with

$$\mathcal{B}_j = \mathcal{L}_0(s) \sqcup \ldots \mathcal{L}_{j-1}(s), \quad \mathcal{S}_j = \mathcal{L}_j(s) \quad \text{and} \quad \mathcal{W}_j = \mathcal{L}_{j+1}(s) \sqcup \ldots \mathcal{L}_r(s), \quad 1 \le j \le k-1. \tag{4.1}$$

We move any redundant vertices from $\mathcal{S}_j$ into $\mathcal{B}_j$, evaluate the partition (using one of the cost functions (2.4) or (2.5)), and choose the one with the minimum cost. The single root level set approach is summarised as Algorithm 3 (where $\mathsf{cost} = \mathsf{cost1}$ or $\mathsf{cost2}$).

Algorithm 3 Level set partitioning algorithm

`function` $\phi(\mathcal{B},\mathcal{W},\mathcal{S}) =$ `LevelSetAlg`$(\mathcal{V},\mathcal{E})$

$(s,t) =$ `PseudoPairAlg`$(\mathcal{V},\mathcal{E})$.
Construct the level-set structure $\mathcal{L}(s)$ of depth k.
Initialise $cost = \infty$
for $j = 1, k-1$ **do**
  Construct the sets $\mathcal{B}_j$, S_j and $\mathcal{W}_j$ given by (4.1)
  Move redundant vertices from $\mathcal{S}_j$ into $\mathcal{B}_j$
  if $\mathsf{cost}(\mathcal{B}_j,\mathcal{W}_j,\mathcal{S}_j) < cost$ **then**
    $cost = \mathsf{cost}(\mathcal{B}_j,\mathcal{W}_j,\mathcal{S}_j)$
    $\phi = \phi_j$
  end if
end for

4.2 Half-level set approach

We found in our experiments that the single root level set approach can sometimes fail to provide a good initial partitioning and so we now propose an alternative level-set based scheme. This uses two level-set structures, $\mathcal{L}(s)$ and $\mathcal{L}(t)$. Assume that $\mathcal{L}(s)$ has k_s+1 level sets numbered from 0 to k_s and $\mathcal{L}(t)$ has k_t+1 level sets numbered from 0 to k_t (note that $k_s = k_t = k$ if s and t are the endpoints of the same pseudo-diameter). For $i = -k_t, \ldots, k_s$, we define the sets

$$\mathcal{S}_i = \{w \in \mathcal{V} \mid i = dist(s,w) - dist(t,w)\}.$$

Equivalently, $\mathcal{S}_i$ can be defined by

$$\mathcal{S}_i = \bigsqcup_{r=\max(0,i)}^{\min(k_s,k_t+i)} \{\mathcal{L}_r(s) \cap \mathcal{L}_q(t) \mid q = r-i\}\,. \tag{4.2}$$

We have the following results for the sets $\mathcal{S}_i$.

Lemma 4.1 Given $\mathcal{G} = (\mathcal{V},\mathcal{E})$, if $(x,y) \in \mathcal{E}$ with $x \in \mathcal{S}_i$ and $y \in \mathcal{S}_j$, then $|i-j| \le 2$.

Proof. Assume $x \in \mathcal{L}_r(s)$ then, since $(x,y) \in \mathcal{E}$, y must belong to $\mathcal{L}_j(s)$ for $j = r-1, r$ or $r+1$. Similarly, assuming $x \in \mathcal{L}_q(t)$, y must belong to $\mathcal{L}_j(t)$ for $j = q-1, q$ or $q+1$. If $x \in \mathcal{L}_r(s)$ and $x \in \mathcal{L}_q(t)$ then, from the definition, $x \in \mathcal{S}_i$ with $i = r-q$. If $y \in \mathcal{S}_j$, then $r-q-2 \le j \le r-q+2$. It follows that $-2 \le i-j \le 2$. □

Lemma 4.2 For all $i = -k_t+1, \ldots, k_s-2$, the set $\mathcal{S}_i \sqcup \mathcal{S}_{i+1}$ is a vertex separator of $\mathcal{G} = (\mathcal{V},\mathcal{E})$.

Proof. To prove that $\mathcal{S} = \mathcal{S}_i \sqcup \mathcal{S}_{i+1}$ is a vertex separator, it is necessary to show there exist $\mathcal{B}$ and $\mathcal{W}$ such that (a) $\mathcal{B}$, $\mathcal{W}$ and $\mathcal{S}$ are non-empty and disjoint sets, (b) $\mathcal{B} \sqcup \mathcal{W} \sqcup \mathcal{S} = \mathcal{V}$ and (c) if $x \in \mathcal{B}$ and $y \in \mathcal{W}$, then $(x,y) \notin \mathcal{E}$. Let $\mathcal{B} = \bigsqcup_{j=-k_t}^{i-1} \mathcal{S}_j$ and $\mathcal{W} = \bigsqcup_{j=i+2}^{k_s} \mathcal{S}_j$. Clearly, (a) and (b) are satisfied. If

$x \in \mathcal{B}$ and $y \in \mathcal{W}$, then $x \in \mathcal{S}_r$ for some $r \in \{-k_t \ldots, i-1\}$ and $y \in \mathcal{S}_q$ for some $q \in \{i+2, \ldots, k_s\}$. Hence, $|r-q| \geq 3$ and, from Lemma 4.1, $(x,y) \notin \mathcal{E}$ and (c) is satisfied. □

From Lemma 4.2, any of the sets $\mathcal{S} = \mathcal{S}_i \sqcup \mathcal{S}_{i+1}$ $(i = 1-l, \ldots, k_s - 2)$ may be selected as the vertex separator, with $\mathcal{B} = \bigsqcup_{j=-k_t}^{i-1} \mathcal{S}_j$ and $\mathcal{W} = \bigsqcup_{j=i+2}^{k_s} \mathcal{S}_j$. We choose i to minimise $\mathsf{cost}(\mathcal{B}, \mathcal{W}, \mathcal{S})$. Our half-level set partitioning algorithm is summarised as Algorithm 4. Note that s and t are chosen to be the endpoints of a pseudo-diameter (although Lemma 4.2 is more general).

Algorithm 4 Half-level set partitioning algorithm

`function` $\phi(\mathcal{B}, \mathcal{W}, \mathcal{S}) =$ `HalfLevelAlg`$(\mathcal{V}, \mathcal{E})$

$(s,t) =$ `PseudoPairAlg`$(\mathcal{V}, \mathcal{E})$.
Construct the rooted level-set structures $\mathcal{L}(s)$ and $\mathcal{L}(t)$ of depth k.
For $i = -k, \ldots, k$, construct the sets $\mathcal{S}_i$ given by (4.2).
Initialise $cost = \infty$
for $i = 1-k, k-2$ **do**
  Set $\mathcal{S}^i = \mathcal{S}_i \sqcup \mathcal{S}_{i+1}$, $\mathcal{B}^i = \bigsqcup_{j=-k}^{i-1} \mathcal{S}_j$ and $\mathcal{W}^i = \bigsqcup_{j=i+2}^{k} \mathcal{S}_j$.
  if $\mathsf{cost}(\mathcal{B}^i, \mathcal{W}^i, \mathcal{S}^i) < cost$ **then**
    $cost = \mathsf{cost}(\mathcal{B}^i, \mathcal{W}^i, \mathcal{S}^i)$
    $\phi(\mathcal{B}, \mathcal{W}, \mathcal{S}) = \phi(\mathcal{B}^i, \mathcal{W}^i, \mathcal{S}^i)$
  end if
end for

5 Multilevel approach

Since the 1990s, multilevel techniques have been widely used for finding separators. They have proved capable of finding high quality separators efficiently. The main objective of a multilevel algorithm is to create a hierarchy of graphs, each representing the original graph, but with a smaller dimension. The smallest (that is, the coarsest) graph in the sequence is partitioned. This partition is propagated back through the sequence of graphs, while being periodically refined.

There are a number of ways to coarsen an undirected graph. The most popular is based on edge collapsing [19, 26] in which pairs of adjacent vertices are selected and each pair is coalesced into a single new vertex with vertex weight equal to the sum of the weights of the two vertices. If multiple edges connect the same pair of vertices, they are replaced by a single edge with an edge weight equal to the sum of the weights of the edges it replaces. We offer two algorithms for edge collapsing: sorted heavy-edge matching and common neighbour matching. Both are greedy algorithms. Sorted heavy-edge matching preferentially collapses heavier edges examining vertices in order of ascending degree, while common neighbour matching coalesces a vertex with its neighbouring vertex that shares the highest number of common neighbours with it. Note that, following [7, 24, 35], we performed early experiments using a maximal independent vertex set strategy but found that this generally yielded orderings that were of poorer quality.

To outline the basic multilevel algorithm, we use subscripts f and c to represent fine and coarse graph quantities, respectively. For example, $\mathcal{G}_f(\mathcal{V}_f, \mathcal{E}_f)$ denotes the fine graph with n_f vertices and $\mathcal{G}_c$ is the graph with n_c vertices obtained after coarsening $(0.5n_f \leq n_c < n_f)$. We associate with $\mathcal{G}_f$ an $n_f \times n_f$ adjacency matrix G_f with a nonzero entry in position (i,j) if and only if vertices i and j are adjacent in $\mathcal{G}_f$. G_c is defined analogously.

When moving from a coarse graph to a fine graph, the partition on the coarse graph must be mapped onto the fine graph. This mapping is represented by a prolongation (or interpolation) matrix. The prolongation step injects the position of a vertex j in the coarse graph to give a value to its parent (or parents). A parent of j is defined to be a vertex in the fine graph that either coalesces into j, or remains as j itself. The prolongation matrix P thus has entries P_{ij} given by

$$P_{ij} \;=\; \begin{cases} 1, & \text{if fine graph vertex } i \text{ is a parent of coarse graph vertex } j, \\ 0, & \text{otherwise,} \end{cases}$$

and the coarse graph may be expressed as the Galerkin product

$$G_c \;\leftarrow\; P^T\, G_f\, P.$$

The coarse grid partition ϕ_c is prolonged onto the fine grid by the prolongation operation

$$\phi_f \;=\; P\phi_c.$$

The finer grid partition ϕ_f may then be improved using the techniques discussed in Section 6. Note that for simplicity of notation, here and elsewhere, we are assuming that the vertices of each graph have been locally renumbered as $1, 2, 3, ...$

Let `CoarsestAlg` be the routine that returns the partition on the coarsest graph (in `SPRAL_ND` this is either `LevelSetAlg` or `HalfLevelAlg`) and let `RefineAlg` denote the routine that takes an initial partition ϕ and returns a refined partition (see Algorithm 6 in Section 6). With this notation, the multilevel partitioning algorithm is summarised as Algorithm 5. Coarsening terminates when n_c is smaller than a

Algorithm 5 Multilevel algorithm

`recursive function` $\phi_f =$ `MultilevelAlg`$(\mathcal{V}_f, \mathcal{E}_f)$

if coarsening has terminated **then**
  $\phi_f^1 =$ `CoarsestAlg`$(\mathcal{V}_f, \mathcal{E}_f)$
else
  Set up the prolongation matrix P
  Construct $G_c \;\leftarrow\; P^T\, G_f\, P$ and $\mathcal{G}_c(\mathcal{V}_c, \mathcal{E}_c)$
  $\phi_c =$ `MultilevelAlg`$(\mathcal{V}_c, \mathcal{E}_c)$
  $\phi_f^1 \;=\; P\phi_c$
end if
$\phi_f =$ `RefineAlg`(ϕ_f^1)

prescribed threshold, or the number of levels exceeds a chosen limit, or the coarsening becomes too slow (the reduction in the number vertices between two levels is not sufficiently large), or it is too rapid (the reduction between two levels is too great). Based on our experiments, in `SPRAL_ND` the coarse graph size in the multilevel hierarchy below which no further coarsening is performed is 100, the maximum number of levels in the multilevel hierarchy is 20, and the maximum and minimum graph reduction factors are 0.9 and 0.5, respectively. In practice, the coarsening process normally terminates because the graph size is smaller than the prescribed threshold; occasionally, coarsening becomes too slow but it is rare that the limit on the number of levels is exceeded.

We have found that it is important to get a good quality partition on the coarsest graph because, if we do not, the refinement algorithm may be unable to produce a good partition at higher levels in the multilevel hierarchy, or will require excessive time to do this.

6 Improving and refining a partition

In this section, we discuss ways in which we can improve a partition $\phi(\mathcal{B}, \mathcal{W}, \mathcal{S})$ in the sense of obtaining a new partition with a better value of the cost function as defined in Section 2. There are many ways to do this; here we limit our attention to those that are currently available within `SPRAL_ND`. The widely used Fiduccia-Mattheyses algorithm (see Algorithm 11 in Section 6.2) for improving a partition moves one vertex at a time. However, sometimes a more dramatic improvement can be obtained by expanding the separator $\mathcal{S}$ to a wide separator. We use the simplest possible method for expanding the separator $\mathcal{S}$: for each $v \in \mathcal{S}$, we include all the neighbours of v. This is very straightforward to implement and quickly creates a wide separator. In early experiments we tried more complicated algorithms for expanding the separator but found that, although we were able to obtain better wide separators, the extra time required was not justified by the limited gains in the quality of the final ordering. Once we have a wide separator, we reduce it to a minimal separator. We then move back and forth between a minimal separator and a wide separator, keeping track of the best partition we have found so far. We terminate this cycle if the maximum number of cycles has been reached or if we fail to improve the partition after a cycle. Algorithm 6 describes our overall refinement strategy `RefineAlg`. Here `MinSepAlg` denotes the techniques described in Section 6.1 that are used to reduce a wide separator to a minimal separator and `FiducciaMattheysesAlg` denotes the Fiduccia-Mattheyses algorithm. Note that `RefineAlg` starts by using `MinSepAlg` since the input separator $\mathcal{S}$ may be a wide separator, for instance if it has come from the half-level set partitioning algorithm.

Algorithm 6 Refinement algorithm

`function` $\phi(\mathcal{B}^1, \mathcal{W}^1, \mathcal{S}^1) =$ `RefineAlg`$(\phi(\mathcal{B}, \mathcal{W}, \mathcal{S}))$

$\phi(\mathcal{B}^1, \mathcal{W}^1, \mathcal{S}^1) =$ `MinSepAlg`$(\phi(\mathcal{B}, \mathcal{W}, \mathcal{S}))$
while maximum number of cycles not reached **do**
  Expand separator $\mathcal{S}^1$ to give the partition $\phi(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}})$
  $\phi(\widehat{\mathcal{B}}, \widehat{\mathcal{W}}, \widehat{\mathcal{S}}) =$ `MinSepAlg`$(\phi(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}}))$
  if `cost`$(\widehat{\mathcal{B}}, \widehat{\mathcal{W}}, \widehat{\mathcal{S}}) \leq$`cost`$(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}})$ **then**
    Set $\phi(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}}) = \phi(\widehat{\mathcal{B}}, \widehat{\mathcal{W}}, \widehat{\mathcal{S}})$
  else
    Exit loop
  end if
  $\phi(\mathcal{B}^1, \mathcal{W}^1, \mathcal{S}^1) =$ `FiducciaMattheysesAlg`$(\phi(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}}))$
end while

6.1 Creating a minimal separator from a wide separator

The approach that algorithm `MinSepAlg` uses to reduce a wide separator to a minimal separator depends on the properties of the input partition ($\mathcal{B}$, $\mathcal{W}$, $\mathcal{S}$) and the balance parameter α (Section 2). If `imb` $< \alpha$ then the separator is trimmed using either `BlockTrimAlg`, a block trimming algorithm (Section 6.1.1) or `FineTrimAlg`, a fine trimming algorithm (Section 6.1.2): the choice of which is used is dependent on the value of $\min(|\mathcal{B}|, |\mathcal{W}|) + |\mathcal{S}|$ relative to $\max(|\mathcal{B}|, |\mathcal{W}|)$, see Algorithm 7. Both of the trimming algorithms return a separator that is a subset of the input separator. If `imb` $\geq \alpha$ we assume, without loss of generality, that $|\mathcal{B}| > |\mathcal{W}|$, and we *shift* the separator $\mathcal{S}$ to form a new wide separator $\widetilde{\mathcal{S}}$, which comprises vertices u and v, where $(u, v) \in \mathcal{E}$ with $u \in \mathcal{S}$ and $v \in \mathcal{B}$, and the corresponding new partition $\phi(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}})$ is then

formed; the maxflow algorithm (Section 6.1.3) is then applied to $\widetilde{\mathcal{S}}$ to return a minimal separator. The algorithm `MinSepAlg` is described in Algorithm 7.

Algorithm 7 Create minimal separator

`function` $\phi(\mathcal{B}^1, \mathcal{W}^1, \mathcal{S}^1) =$ `MinSepAlg`$(\phi(\mathcal{B}, \mathcal{W}, \mathcal{S}))$

if `imb` $< \alpha$ **then**
  if $\min(|\mathcal{B}|, |\mathcal{W}|) + |\mathcal{S}| < \max(|\mathcal{B}|, |\mathcal{W}|)$ **then**
    $\phi(\mathcal{B}^1, \mathcal{W}^1, \mathcal{S}^1) =$ `BlockTrimAlg`$(\phi(\mathcal{B}, \mathcal{W}, \mathcal{S}))$
  else
    $\phi(\mathcal{B}^1, \mathcal{W}^1, \mathcal{S}^1) =$ `FineTrimAlg`$(\phi(\mathcal{B}, \mathcal{W}, \mathcal{S}))$
  end if
else
  if $|\mathcal{B}| < |\mathcal{W}|$ **then**
    $\widetilde{\mathcal{S}} = \{u, v | (u, v) \in \mathcal{E}, u \in \mathcal{S}, v \in \mathcal{W}\}$, $\widetilde{\mathcal{B}} = \mathcal{B} \sqcup \left(\mathcal{S} \setminus \left(\widetilde{\mathcal{S}} \cap \mathcal{S}\right)\right)$, $\widetilde{\mathcal{W}} = \mathcal{W} \setminus \widetilde{\mathcal{S}}$
  else
    $\widetilde{\mathcal{S}} = \{u, v | (u, v) \in \mathcal{E}, u \in \mathcal{S}, v \in \mathcal{B}\}$, $\widetilde{\mathcal{B}} = \mathcal{B} \setminus \widetilde{\mathcal{S}}$, $\widetilde{\mathcal{W}} = \mathcal{W} \sqcup \left(\mathcal{S} \setminus \left(\widetilde{\mathcal{S}} \cap \mathcal{S}\right)\right)$,
  end if
  $\phi(\mathcal{B}^1, \mathcal{W}^1, \mathcal{S}^1) =$ `MaxflowAlg`$(\phi(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}}))$
end if

6.1.1 Block Trimming

Assume that the separator $\mathcal{S}$ in the partition $\phi(\mathcal{B}, \mathcal{W}, \mathcal{S})$ is not minimal so that there is at least one vertex in $\mathcal{S}$ that is not connected to vertices in both $\mathcal{B}$ and $\mathcal{W}$. Block trimming moves subsets of vertices from $\mathcal{S}$ to either $\mathcal{B}$ or $\mathcal{W}$ in order to get a minimal separator. We do this by finding all vertices in $\mathcal{S}$ that are connected to vertices in $\mathcal{B}$ but are not connected to any vertices in $\mathcal{W}$ and computing the cost, $cost_{\mathcal{B}}$, of the partition that would result from moving all of these vertices into $\mathcal{B}$. Similarly, we find all vertices in $\mathcal{S}$ that are connected to vertices in $\mathcal{W}$ but are not connected to any vertices in $\mathcal{B}$ and compute the cost, $cost_{\mathcal{W}}$, of the partition that would result from moving all of these vertices were into $\mathcal{W}$. If $cost_{\mathcal{B}} < cost_{\mathcal{W}}$, the vertices in $\mathcal{S}$ that are connected to vertices in $\mathcal{B}$ but not connected to any vertices in $\mathcal{W}$ are moved into $\mathcal{B}$; otherwise, the vertices in $\mathcal{S}$ that are connected to vertices in $\mathcal{W}$ but not connected to any vertices in $\mathcal{B}$ are moved into $\mathcal{W}$. The partition is updated and this process repeated until no further vertices can be moved in this manner. Finally, we check for any vertices in $\mathcal{S}$ that are only connected to vertices that are also in $\mathcal{S}$. If there are such vertices, we move them one at a time into either $\mathcal{B}$ or $\mathcal{W}$, choosing the move that results in a lower cost partition. Algorithm 8 contains the details of the block trimming algorithm `BlockTrimAlg`.

6.1.2 Fine Trimming

Block trimming can move a large number of vertices from $\mathcal{S}$ into either $\mathcal{B}$ or $\mathcal{W}$. In comparison, fine trimming only moves one vertex at a time and is less likely to form a partition that violates `imb` $\geq \alpha$. However, it is more expensive than block trimming so we use block trimming in Algorithm 7 when we expect that it will produce a partition similar to that obtained using fine trimming.

In the fine trimming method, we find a vertex u in $\mathcal{S}$ that is connected to vertices in $\mathcal{B}$ but not connected to any vertices in $\mathcal{W}$ and find a vertex v in $\mathcal{S}$ that is connected to vertices in $\mathcal{W}$ but not connected to any

Algorithm 8 Block trimming

`function` $\phi(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}}) = \texttt{BlockTrimAlg}(\phi(\mathcal{B}, \mathcal{W}, \mathcal{S}))$

Set $\widetilde{\mathcal{B}} = \mathcal{B}$, $\widetilde{\mathcal{W}} = \mathcal{W}$ and $\widetilde{\mathcal{S}} = \mathcal{S}$,

for **do**

Initialise $cost_{\widetilde{\mathcal{B}}} = \infty$, $cost_{\widetilde{\mathcal{W}}} = \infty$

if there are vertices $\widetilde{\mathcal{S}}_{\widetilde{\mathcal{B}}}$ in $\widetilde{\mathcal{S}}$ connected to $\widetilde{\mathcal{B}}$ but not to $\widetilde{\mathcal{W}}$ **then**

Evaluate $cost_{\widetilde{\mathcal{B}}} = \texttt{cost}(\widetilde{\mathcal{S}}_{\widetilde{\mathcal{B}}} \sqcup \widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}} \setminus \widetilde{\mathcal{S}}_{\widetilde{\mathcal{B}}})$

end if

if there are vertices $\widetilde{\mathcal{S}}_{\widetilde{\mathcal{W}}}$ in $\widetilde{\mathcal{S}}$ connected to $\widetilde{\mathcal{W}}$ but not to $\widetilde{\mathcal{B}}$ **then**

Evaluate $cost_{\widetilde{\mathcal{W}}} = \texttt{cost}(\widetilde{\mathcal{B}}, \widetilde{\mathcal{S}}_{\widetilde{\mathcal{W}}} \sqcup \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}} \setminus \widetilde{\mathcal{S}}_{\widetilde{\mathcal{W}}})$

end if

if Both $cost_{\widetilde{\mathcal{B}}}$ and $cost_{\widetilde{\mathcal{W}}}$ equal ∞ **then**

exit

end if

if $cost_{\widetilde{\mathcal{B}}} < cost_{\widetilde{\mathcal{W}}}$ **then**

Move $\widetilde{\mathcal{S}}_{\widetilde{\mathcal{B}}}$ from $\widetilde{\mathcal{S}}$ to $\widetilde{\mathcal{B}}$

else

Move $\widetilde{\mathcal{S}}_{\widetilde{\mathcal{W}}}$ from $\widetilde{\mathcal{S}}$ to $\widetilde{\mathcal{W}}$

end if

end for

while there exists $u \in \widetilde{\mathcal{S}}$ not connected to $\widetilde{\mathcal{B}}$ or $\widetilde{\mathcal{W}}$ **do**

Move u to $\widetilde{\mathcal{B}}$ or $\widetilde{\mathcal{W}}$ depending on which move results in the smaller cost function.

end while

vertices in $\mathcal{B}$. We compute the cost, $cost_{\mathcal{B}}$, of the partition that would result from moving u into $\mathcal{B}$ and the cost $cost_{\mathcal{W}}$, of the partition that would result from moving v into $\mathcal{W}$. If $cost_{\mathcal{B}} < cost_{\mathcal{W}}$, u is moved into $\mathcal{B}$; otherwise, v is moved into $\mathcal{W}$. The partition is updated and we repeat the process until each vertex in $\mathcal{S}$ that is connected to a vertex in $\mathcal{B}$ is also connected to a vertex in $\mathcal{W}$. Finally, any vertices in $\mathcal{S}$ that are only connected to other vertices in $\mathcal{S}$ are moved out of $\mathcal{S}$, as in the block trimming method. The fine trimming algorithm `FineTrimAlg` is outlined in Algorithm 9.

Algorithm 9 Fine trimming

`function` $\phi(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}}) =$ `FineTrimAlg`$(\phi(\mathcal{B}, \mathcal{W}, \mathcal{S}))$

Set $\widetilde{\mathcal{B}} = \mathcal{B}$, $\widetilde{\mathcal{W}} = \mathcal{W}$ and $\widetilde{\mathcal{S}} = \mathcal{S}$,

finemove = .true.

while finemove **do**

Initialise finemove = .false., $cost_{\widetilde{\mathcal{B}}} = \infty$, $cost_{\widetilde{\mathcal{W}}} = \infty$

if there exists a vertex $v_{\widetilde{\mathcal{B}}}$ in $\widetilde{\mathcal{S}}$ connected to $\widetilde{\mathcal{B}}$ but not to $\widetilde{\mathcal{W}}$ **then**

Evaluate $cost_{\widetilde{\mathcal{B}}} =$ `cost`$(\{v_{\widetilde{\mathcal{B}}}\} \sqcup \widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}} \setminus \{v_{\widetilde{\mathcal{B}}}\})$

Set finemove = .true.

end if

if there exists a vertex $v_{\widetilde{\mathcal{W}}}$ in $\widetilde{\mathcal{S}}$ connected to $\widetilde{\mathcal{W}}$ but not to $\widetilde{\mathcal{B}}$ **then**

Evaluate $cost_{\widetilde{\mathcal{W}}} =$ `cost`$(\widetilde{\mathcal{B}}, \{v_{\widetilde{\mathcal{W}}}\} \sqcup \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}} \setminus \{v_{\widetilde{\mathcal{W}}}\})$

Set finemove = .true.

end if

if $cost_{\widetilde{\mathcal{B}}} < cost_{\widetilde{\mathcal{W}}}$ **then**

Move $v_{\widetilde{\mathcal{B}}}$ from $\widetilde{\mathcal{S}}$ to $\widetilde{\mathcal{B}}$

else

Move $\widetilde{\mathcal{S}}_{\widetilde{\mathcal{W}}}$ from $\widetilde{\mathcal{S}}$ to $\widetilde{\mathcal{W}}$

end if

end while

while there exists $u \in \widetilde{\mathcal{S}}$ not connected to $\widetilde{\mathcal{B}}$ or $\widetilde{\mathcal{W}}$ **do**

Move u to $\widetilde{\mathcal{B}}$ or $\widetilde{\mathcal{W}}$ depending on which move results in the smaller cost function.

end while

6.1.3 Maxflow

We construct and solve a maxflow problem to improve the separator and use the Ford-Fulkerson algorithm [28] to solve this maxflow problem. The central structure for this approach is a network graph, and we follow the work of [5] by defining the network $\mathcal{N}$ in terms of a node set and a set of arcs. All vertices in the set $\mathcal{B}$ are considered as a single source node s and those in $\mathcal{W}$ as a single sink node t. There are arcs in $\mathcal{N}$ from s to all vertices in the separator $\mathcal{S}$ that are connected by an edge to a vertex in $\mathcal{B}$. Similarly there are arcs from t to all vertices in $\mathcal{S}$ that are connected to vertices in $\mathcal{W}$. Each vertex in $\mathcal{S}$ is split into two nodes connected by an arc of the network. We designate the splitting of vertex u by the two nodes u^- and u^+ connected by an arc. Any incoming edge (v, u) goes to u^- and any outgoing edge (u, w) comes from u^+. The crucial aspect is that the capacity on the arcs (u^-, u^+) is defined by the vertex weight whereas all other arcs are considered to have infinite capacity. This is illustrated in Figure 6.1.

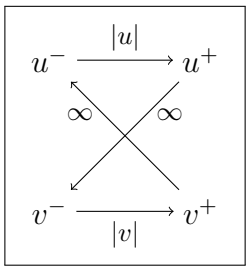

Figure 6.1: Node structure in network graph

We use the Ford-Fulkerson algorithm on this network graph to find the maximum flow between source and sink. The algorithm is a simple greedy algorithm that keeps trying to augment the flows subject to the capacity limits until further increase is no longer possible. We do this through a breadth-first search of $\mathcal{N}$ starting at s although a depth first search is equally possible. This algorithm has complexity bounded by the value of the maximum flow times the number of arcs in the network.

The Ford-Fulkerson theorem indicates that the minimum cut in the network will have value equal to the maximum flow. The cut set need not be unique even though its value is. We scan the network from s to find a minimum cut and scan from t to find another. The cut with the better value for our cost function is used to define the new separator set. This is summarised as Algorithm 10.

Algorithm 10 Maxflow Algorithm

`function` $\phi(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}}) =$ `MaxflowAlg`$(\phi(\mathcal{B}, \mathcal{W}, \mathcal{S}))$

Shrink $\mathcal{B}$ to source node s
Shrink $\mathcal{W}$ to sink node t
Construct network graph $\mathcal{N}$ with s, t and $\mathcal{S}$
Use Ford-Fulkerson algorithm to compute maxflow on $\mathcal{N}$
  Initialize flows to zero
  while more arcs from s **do**
    Find augmenting path starting with this arc and finishing at t
    Update flows
    Continue until no further augmenting paths can be found
  end while
Find minimum cut starting from s using a breadth first search
Find minimum cut starting from t using a breadth first search
Evaluate the two cuts and keep the one with lesser cost
Update partition $\phi(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}})$ so that $\widetilde{\mathcal{S}}$ corresponds to this minimum cut

One issue with the maxflow algorithm as presently described is that it does not try to obtain a balanced partition. We are just given two choices for a minimal separator and choose the better of the two. There may, however, be other separators lying between these two extremes. We try to address this issue by using *augmented capacities* where the balance of the partition is included in the edge weights of our network graph. We do this by constructing a half-level set structure for the vertices in $\mathcal{S}$ (that are represented by two nodes in the network graph). We assign a value to each vertex ($penalty(s)$) that is based on the imbalance given by the half-level set in which the vertex lies. The augmented capacities for each edge of the form (u^-, u^+)) are then given by

$$capacity(u^-, u^+) = \lceil 100 * penalty(s) * |s| \rceil$$

6.2 Fiduccia-Mattheyses

The Fiduccia-Mattheyses algorithm [15] may be used to refine a partition $\phi(\mathcal{B}, \mathcal{W}, \mathcal{S})$. The algorithm in the original paper was defined for edge separators but it is easy to express this in terms of vertex separators. The algorithm then makes small moves, one vertex at a time. This vertex implementation was first done by Ashcraft and Liu [4], who call these "primitive moves".

The algorithm consists of two loops. Each iteration of the inner loop generates a potential new partition. If this potential partition is the best found so far in the inner loop, the potential partition is updated to this and the inner loop continues to the next iteration.

In each iteration of the inner loop, a vertex j from the separator $\mathcal{S}$ is chosen; a notion of *gain* is used to choose this vertex. The gain of a vertex $j \in \mathcal{S}$ is defined to be

$$gain(j) = \min \left\{ gain_{\mathcal{B}}(j), gain_{\mathcal{W}}(j) \right\},$$

where $gain_{\mathcal{B}}(j) = |\mathcal{S}| - |\widetilde{\mathcal{S}}|$ and $\widetilde{\mathcal{S}}$ is the separator that would result from moving j into $\mathcal{B}$ and moving any neighbours of j that are in $\mathcal{W}$ into the separator; $gain_{\mathcal{W}}(j)$ is defined analogously.

Each $j \in \mathcal{S}$ is placed in a bucket according to its gain value so long as the gain value is less than some pre-defined value. In our implementation, we set this to

$$\min(\sum |w_i|, 5 * \max|w_i|),$$

where w_i are the vertex weights. A vertex is chosen from the bucket that contains the vertices with the lowest gains; it is placed into $\mathcal{B}$ or $\mathcal{W}$ depending on which results in the smaller value of the partition evaluation function `cost` given in equation (2.4).

If a vertex has already been moved out of the separator and later re-enters the separator, it cannot be chosen to be moved again, i.e., it cannot be placed in a bucket. Conversely, if a vertex is moved into the separator and has not previously been in the separator, its gain will be calculated and, if less than the pre-defined value, it will be placed in the relevant bucket. Additionally, the gain values of any vertices that are already in the separator and are neighbours of j are updated and moved into the appropriate bucket. The inner loop terminates when there are no vertices left in the gain buckets.

If the cost of the partition found after exiting the inner loop is not smaller than that of the current best partition, the algorithm terminates. If the cost has decreased, the inner loop is called again starting from the now best partition. The process is described in Algorithm 11.

To improve efficiency, `SPRAL_ND` uses a "band form" of the algorithm in which a vertex can only be moved out of the separator if the vertices that then come into the separator are at most a distance "band" from the initial separator. The concept of using a "band form' is similar to that proposed by Chevalier and Pellegrini [9]. Additionally, to limit the cost of the Fiduccia-Mattheyses algorithm, we always reduce the separator to a minimal separator before applying the algorithm.

Algorithm 11 Fiduccia-Mattheyses Algorithm

`function` $\phi(\mathcal{B}^1, \mathcal{W}^1, \mathcal{S}^1) =$ `Fiduccia`$\mathtt{Mattheyses}$`Alg`$(\mathcal{V}, \mathcal{E}, \phi(\mathcal{B}, \mathcal{W}, \mathcal{S}))$

$cost_c = \mathsf{cost}(\mathcal{B}, \mathcal{W}, \mathcal{S})$
Set $(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}}) = (\mathcal{B}, \mathcal{W}, \mathcal{S})$
Initialise $cost_{best} = \infty$
for do
  Set $\phi(\mathcal{B}^1, \mathcal{W}^1, \mathcal{S}^1) = \phi(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}})$
  Evaluate $gain(j)$ for all $j \in \widetilde{\mathcal{S}}$.
  while there are vertices in $\widetilde{\mathcal{S}}$ that have not been visited in this loop **do**
    Choose vertex of least gain and compute possible new partition and cost
    if $\mathsf{cost} < cost_{best}$ **then**
      $cost_{best} = \mathsf{cost}$
      Set $\phi(\widetilde{\mathcal{B}}, \widetilde{\mathcal{W}}, \widetilde{\mathcal{S}})$ to best partition
    end if
    Determine all $j \in \widetilde{\mathcal{S}}$ for which $gain(j)$ has changed and update these gains
  end while
  if $cost_{best} \geq cost_c$ **then**
    exit
  else
    $cost_c = cost_{best}$
  end if
end for

7 The complete algorithm

Our sparse matrix ordering algorithm, based on nested dissection that is implemented within `SPRAL_ND` is summarised as Algorithm 12. The user must choose whether to use the non-multilevel or multilevel approach in implementing *PartitionAlg* as defined in Section 3.

Algorithm 12 Sparse matrix ordering algorithm

$\pi =$ `SparseOrderAlg`$(A, n, PartitionAlg)$

Remove dense rows/columns from A

Optionally compress the matrix and initialise the vertex weights

Let $\widetilde{A}$ be the resulting matrix

for each independent component $\widetilde{A}_i$ of $\widetilde{A}$ **do**

$\widetilde{\pi}_i =$ `nested_dissection`$(\widetilde{A}_i, \widetilde{n}_i, PartitionAlg)$

end for

Combine the $\widetilde{\pi}_i$ to give an ordering $\widetilde{\pi}$ for $\widetilde{A}$

Uncompress the ordering $\pi \leftarrow \widetilde{\pi}$ and append vertices corresponding to dense rows/columns

8 Numerical experiments

The test problems used in this paper are from the University of Florida Sparse Matrix Collection [12] and are chosen to represent a wide range of sparsity structures: if the matrix is unsymmetric we form a symmetrized version $A + A^T$. We also ensure that our test problems are positive definite by setting each diagonal entry a_{ii}, $i = 1, \ldots, n$, to be

$$a_{ii} = 1.0 + 2 \sum_{j=1}^{n} |a_{ij}|,$$

where n is the order of the test problem. We list the test problems and information about each problem in Table A.1 (Appendix A). The problems are in increasing order of $\%band$, the percentage bandwidth (after the Reverse Cuthill-McKee algorithm [17] has been applied) of the largest independent component in $\widetilde{A}$ (the preprocessed matrix) given as a percentage of n_{maxc}, the order of that component. We test our computed elimination ordering by solving $Ax = b$ using the sparse direct Cholesky solver `HSL_MA87` (Version 2.3.0) [21] from the HSL mathematical software library [23], where the right-hand side b is chosen so that the solution is $x_i = 1$ for all i.

Unless stated otherwise, our experiments are performed using double precision reals on a machine with two Intel E5-2695 v3 fourteen core processors running at 2.3GHz. The gfortran compiler (Version 4.9.3) with options -g -O2 -fopenmp and MKL BLAS (Version 11.2.3.187) are used. `SPRAL_ND` is a serial code but `HSL_MA87` is a parallel code using OpenMP: we set `HSL_MA87` to use 28 threads (one for each core).

For each experiment, we compare the number of entries nz in the Cholesky factor L and the number of floating point operations *nflops* required to compute L, as reported by the analyse phase of `HSL_MA87`. We also compare the wall-clock time (in seconds) to

- compute the elimination ordering ($time\_o$);
- perform the analyse phase of `HSL_MA87` ($time\_a$);
- perform the factorise phase of `HSL_MA87` ($time\_f$);

- perform the solve phase of `HSL_MA87` ($time\_s$);
- the total time $time\_t = time\_o + time\_a + time\_f + time\_s$.

We note that for each problem, recording the timings for multiple runs gave a multi-modal distribution because of the presence of frequency scaling and NUMA effects. For example, $time\_o$ for running `SPRAL_ND` on problem turon_m for 10 different runs was $(1.05, 0.85, 0.73, 1.06, 1.05, 1.05, 1.05, 1.06, 0.73)$. Running more than 100 times demonstrates two distinct peaks centered at 0.73 and 1.05. Clearly taking an average of such a distribution provides little value. Instead, we perform ten runs and report the minimum time, being a good estimator of the (narrow) peak of highest performance (other percentile-based statistics such as the median proved considerably less repeatable). The reported total times are the sum of the least individual times.

In all our tests, `SPRAL_ND` pre-processes the input matrix by removing any dense rows (Section 3.1) and compressing the resulting matrix (Section 3.2) to give a (sometimes) smaller matrix $\widetilde{A}$, see Algorithm 12. Note that, in addition to the figures and tables given in the following subsections, performance profiles are presented in Appendix B. These do not provide information on individual problems but are a useful tool for evaluating performance across the test set.

8.1 Comparison of non-multilevel and multilevel partitioning methods

Our first set of experiments compares the use of the non-multilevel (NM) and multilevel (ML) methods. In the ML tests, the sorted heavy-edge matching strategy is used (Section 5). We compute the initial partition with the half-level set approach (Section 4.2). In Figure 8.1, we consider the effect of $\%band$ on the quality of NM versus ML. In general, for test problems with a small bandwidth, the NM ordering is of similar or better quality than the ML ordering in terms of nz and *nflops*; but as $\%band$ increases, the NM orderings can result in *nflops* being up to two orders of magnitude greater than for the ML orderings. $time\_o$ and $time\_t$ are compared in Figure 8.2. As expected, NM generally takes significantly less time to compute the ordering than ML and, in most cases, although it gives a poorer quality ordering, the total solution time $time\_t$ is less for NM.

In Table 8.1, we consider a subset of our test problems that have been selected to demonstrate differences between the two methods when applied to the same problem. It is important to balance the speed of computing the elimination ordering with the quality of the resulting elimination ordering. For example, consider the problem turon_m. The elimination ordering formed by ML is of significantly higher quality in terms of nz and *nflops*, which results in the factorization time being reduced by a third. However, the time to form this elimination is significantly longer than NM so, if only one factorization and one solution of the matrix is required, it is advantageous (in terms of time) to use NM. If many matrices with the same (or similar) sparsity patterns are to factorized or a large number of solutions for different right-hand sides b are required, then the gains in the factorization and solve times for ML will shift the balance towards its use.

8.2 Comparison of half-level set and level set partitioning methods

Our second set of experiments compares the half-level set (HL) and level set (LS) partitioning methods. Here we use NM for problems 1-49 (those for which $\%band$ is less than 3) and ML for the remainder. Figures 8.3 and 8.4 compare the HL and ML methods. We observe that while there are some problems where HL produces a significantly better quality ordering, this is not the case for all problems and we are unable to predict which method is the better for a given problem. The default within `SPRAL_ND` (and in the remainder of this report) is HL but if a number of similarly structured problems are to be ordered, it may be worthwhile to run both approaches on the first problem and use whichever is better for the others.

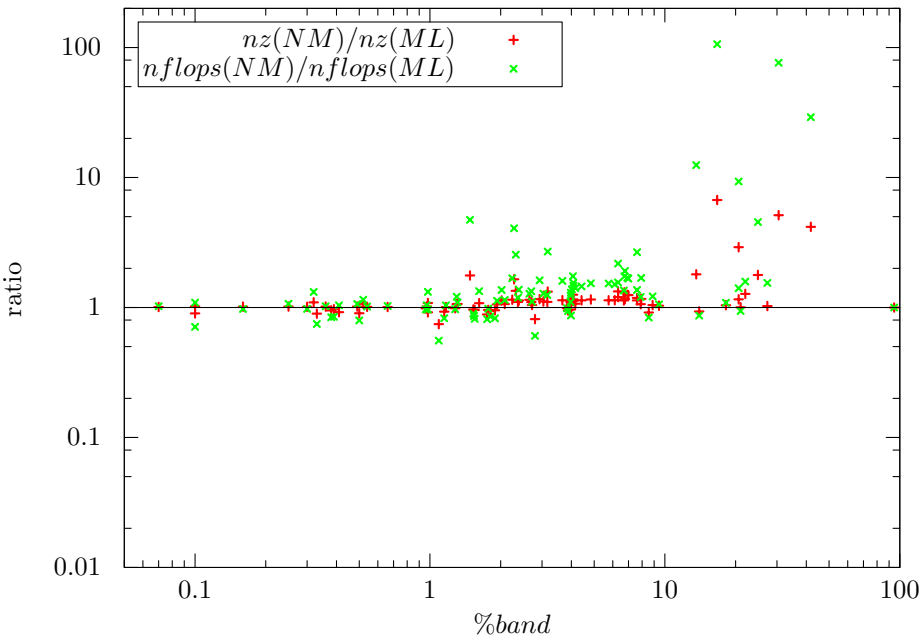

Figure 8.1: The effect of *%band* on *nz* and *nflops* for the non-multilevel (NM) and multilevel (ML) partitioning versions of `SPRAL_ND`.

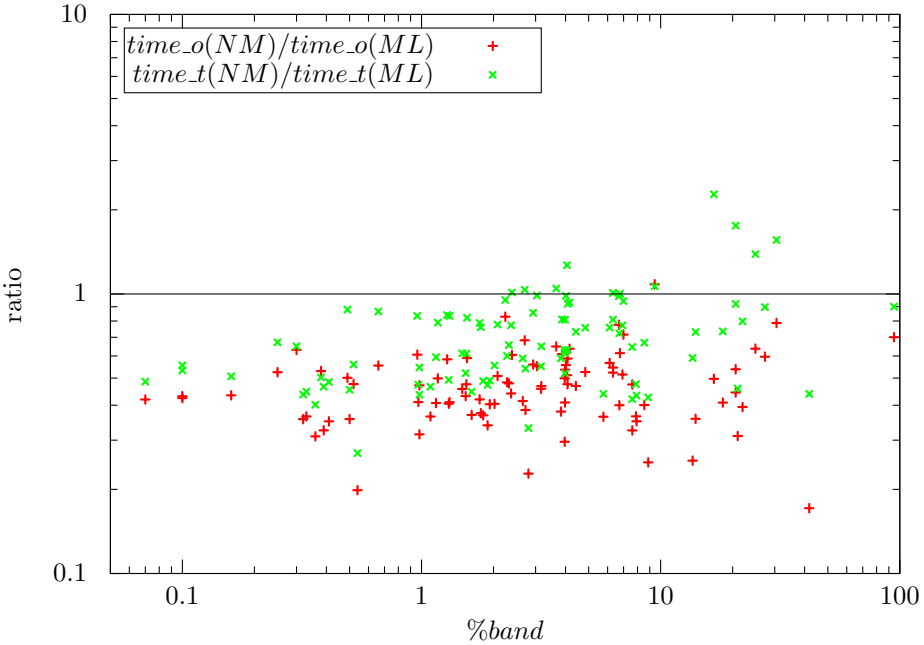

Figure 8.2: The effect of *%band* on *time_o* and *time_t* for the non-multilevel (NM) and multilevel (ML) partitioning versions of `SPRAL_ND`.

Name	n_{maxc}	$\%band$	$nz(10^6)$		$nflops(10^9)$		$time\_o$		$time\_f$		$time\_t$	
			NM	ML	NM	ML	NM	ML	NM	ML	NM	ML
thermal2	1227087	0.07	**104**	**103**	**26.6**	**26.0**	**5.24**	12.5	**1.07**	1.34	**7.19**	14.8
shallow_water1	81920	0.39	**4.86**	**5.01**	**0.52**	0.61	**0.19**	0.58	**0.14**	0.17	**0.37**	0.79
apache2	715176	0.41	**183**	**199**	**210**	**202**	**2.49**	7.11	**1.27**	**1.16**	**4.26**	8.79
ford2	97906	0.98	**6.90**	**6.38**	0.86	**0.66**	**0.29**	0.62	0.05	**0.04**	**0.39**	0.71
halfb	38556	1.55	**93.4**	**97.6**	**108**	132	**0.25**	0.42	**0.55**	0.62	**1.13**	1.38
turon_m	189924	2.28	33.3	**20.2**	21.4	**5.26**	**0.73**	1.51	0.20	**0.13**	**1.05**	1.74
filter3D	106224	3.16	25.5	**23.1**	10.3	**8.25**	**0.64**	1.40	**0.18**	**0.18**	**0.94**	1.70
pkustk10	13446	3.87	**22.3**	**23.5**	**14.0**	**14.9**	**0.07**	0.11	**0.14**	0.17	**0.31**	0.38
audikw_1	314335	4.06	1680	**1340**	11700	**6730**	**3.14**	6.13	21.7	**12.9**	27.8	**21.9**
ct20stif	17723	6.32	16.0	**13.3**	12.5	**8.03**	**0.10**	0.19	0.13	**0.11**	**0.30**	0.37
boneS01	39672	6.32	64.4	**48.4**	116	**53.3**	**0.25**	0.46	0.50	**0.29**	**0.95**	**0.94**
net4-1	5386	8.87	**6.62**	**6.34**	0.76	**0.62**	**0.25**	0.99	0.10	**0.07**	**0.51**	1.19
gupta2	48185	16.7	94.6	**14.1**	594	**5.60**	**0.25**	0.49	2.84	**0.83**	3.77	**1.66**
Andrews	59999	24.9	85.4	**48.0**	342	**75.2**	**0.62**	0.97	1.47	**0.57**	2.39	**1.72**
ins2	56	94.6	**6.73**	**6.73**	**0.21**	**0.21**	**0.12**	0.18	**0.08**	0.10	**0.44**	0.49

Table 8.1: Comparison of the non-multilevel (NM) and multilevel (ML) methods for a subset of test problems. For each problem, the best statistic (and any within 10% of the best) are in bold.

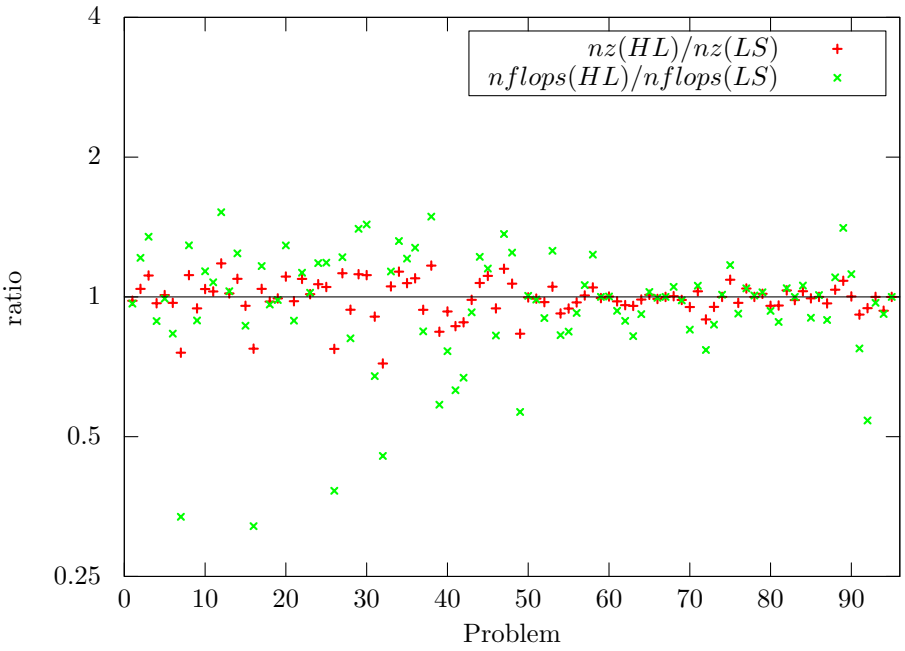

Figure 8.3: Comparison of *nz* and *nflops* for for the half-level set (HL) and level set (LS) partitioning methods in `SPRAL_ND`. The problems are in the order given in Table A.1.

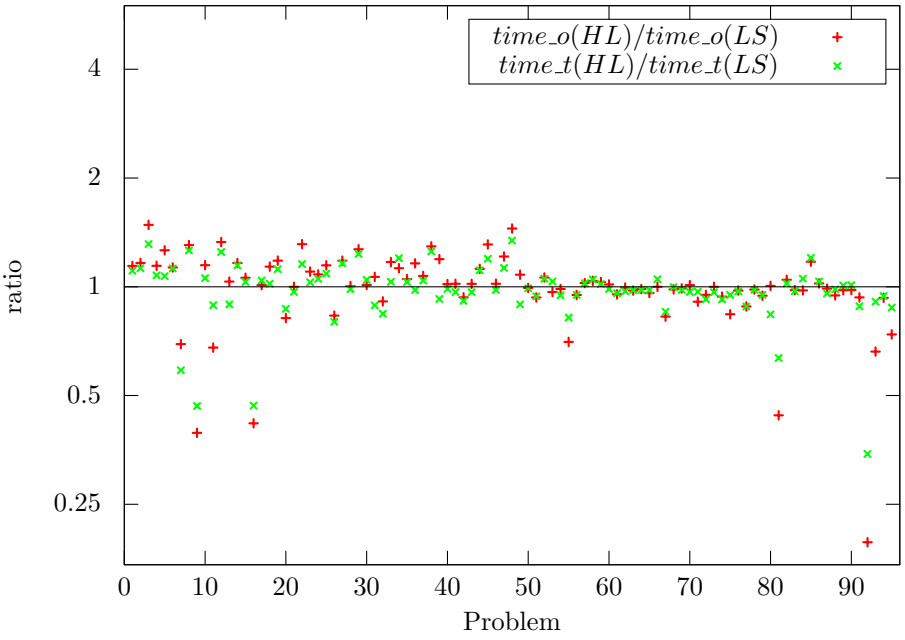

Figure 8.4: Comparison of *time_o* and *time_t* for the half-level set (HL) and level set (LS) partitioning methods in `SPRAL_ND`. The problems are in the order given in Table A.1.

8.3 Comparison of common neighbours and sorted heavy-edge matchings

Our implementation of the multilevel method may use either a common neighbours matching (CNM) strategy or a sorted heavy-edge matching (SHEM) strategy. Figures 8.5 and 8.6 compare the two methods. In terms of quality, SHEM is the best (or joint best) for 83 problems (versus 65 for CNM). In terms of ordering time, neither strategy is clearly better than the other. Table 8.2 gives the details for a subset of problems. As can be seen, the improved ordering quality is generally reflected in the value of *time_f*, though this time is normally dominated by the ordering time. Within `SPRAL_ND`, SHEM is the default strategy and is used in the remainder of this report.

	$nz(10^6)$		$nflops(10^9)$		*time_o*		*time_f*		*time_t*	
Name	CNM	SHEM	CNM	SHEM	CNM	SHEM	CNM	SHEM	CNM	SHEM
onera_dual	**12.4**	**11.4**	5.22	**3.79**	**0.55**	0.64	0.10	**0.09**	**0.70**	0.78
copter2	**14.1**	**14.2**	**8.34**	**8.25**	**0.60**	**0.58**	0.14	**0.12**	**0.79**	**0.76**
dawson5	**5.93**	**6.13**	**1.11**	1.27	**0.23**	0.27	**0.05**	**0.05**	**0.32**	0.36
helm3d01	**7.93**	**7.28**	4.64	**3.59**	0.51	**0.32**	0.09	**0.07**	0.65	**0.43**
CO	**1840**	**1770**	**29000**	**26400**	**4.89**	**5.21**	120	**72.7**	128	**80.4**
GaAsH6	**248**	**252**	**1600**	**1650**	**1.15**	1.28	**4.25**	**4.37**	**5.77**	**6.04**
gupta2	**14.2**	**14.1**	**5.60**	**5.60**	**0.25**	0.47	**0.58**	0.80	**1.16**	1.61
nd12k	**122**	**122**	**562**	**555**	4.98	**1.90**	**1.60**	**1.59**	6.96	**3.86**
ncvxqp7	43.5	**37.1**	100	**61.7**	**1.21**	**1.24**	0.72	**0.51**	2.13	**1.91**
c-56	**2.52**	**2.48**	0.50	**0.42**	**0.34**	**0.32**	**0.04**	**0.04**	**0.43**	**0.41**
gupta1	**3.85**	**3.84**	**0.76**	**0.78**	**0.13**	**0.13**	**0.29**	0.36	**0.60**	0.68
lpl1	3.34	**2.54**	0.79	**0.43**	**0.32**	0.36	0.05	**0.03**	**0.41**	**0.43**

Table 8.2: Comparison of the common neighbours (CNM) and sorted heavy-edge (SHEM) matching strategies for a subset of problems. For each problem, the best statistic (and any within 10% of the best) are in bold.

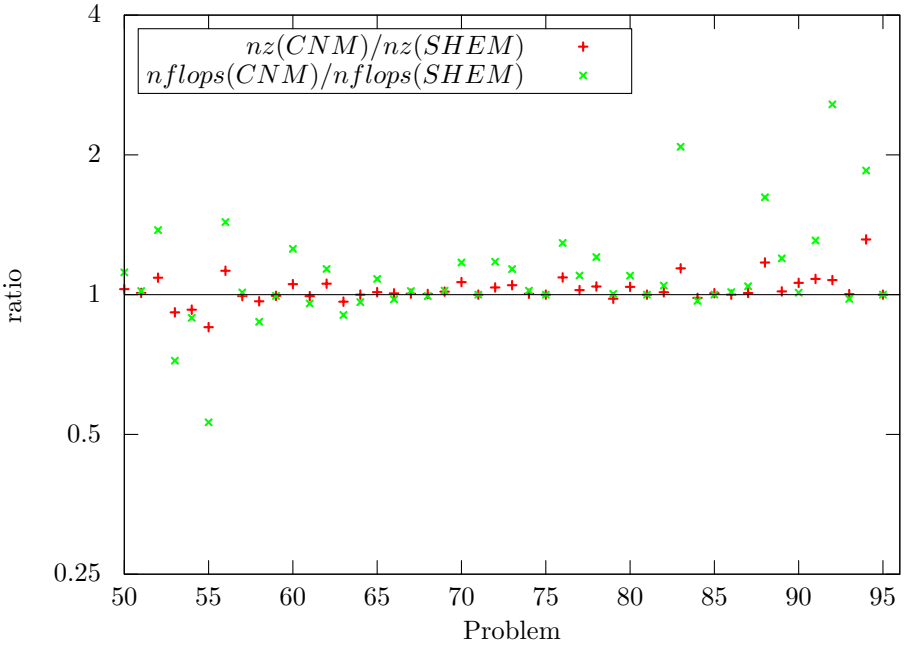

Figure 8.5: Comparison of *nz* and *nflops* for for the common neighbour (CNM) and sorted heavy-edge (SHEM) matching methods in `SPRAL_ND`. The problems are in the order given in Table A.1; problems 1–49 are excluded because they do not use the multilevel scheme.

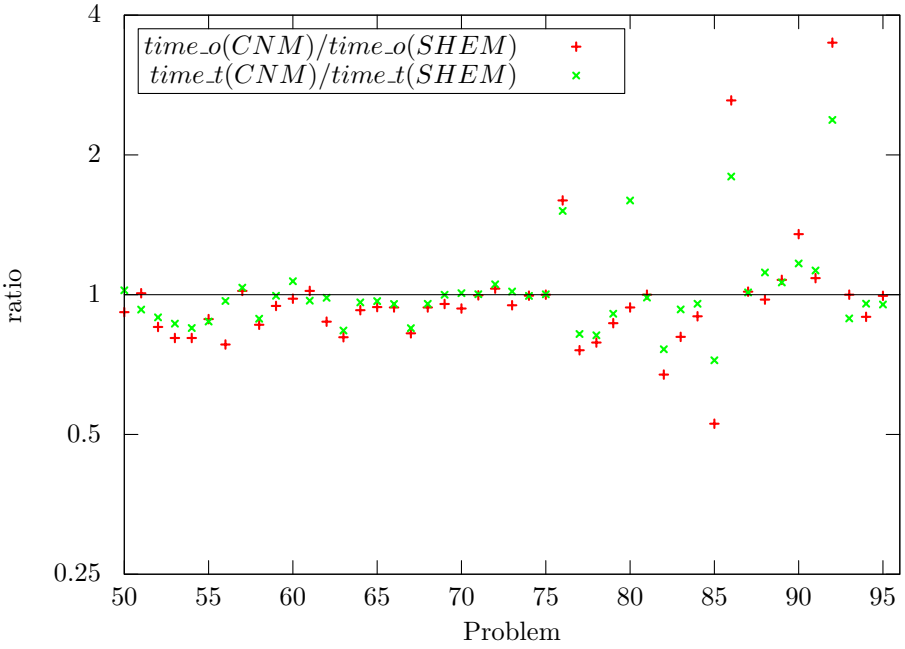

Figure 8.6: Comparison of *time_o* and *time_t* for the common neighbour (CNM) and sorted heavy-edge (SHEM) matching methods in `SPRAL_ND`. The problems are in the order given in Table A.1; problems 1–49 are excluded because they do not use the multilevel scheme.

8.4 Comparison of different values of α

As described in Section 2, the balance of the generated partition can be controlled using the parameter α of equation (2.4). In this section, we compare the values $\alpha = 1.0$, 1.5, 2.0, 3.0, 4.0 and 5.0. In the choice of the balance parameter, there is a tension between two desired outcomes. As α increases, the constraint on the choice of separators is relaxed, allowing a smaller a separator to be chosen (though there is no guarantee it will remain a smaller separator as it is refined through the multilevel scheme). However, the lack of balance can also be detrimental to the quality of the ordering if it results in a very small set $\mathcal{B}$ or $\mathcal{W}$. This tension can be seen in the apparent sweet spot of $\alpha = 4.0$ demonstrated by the ratios shown in Figure 8.7, though the range of variation is relatively small.

With reference to the performance profiles in Appendix B, and Table 8.3, we see that $\alpha = 1.0$ is perhaps better than $\alpha = 4.0$, however the slightly improved quality is offset by an order of magnitude slower ordering times. This slower time is because the method struggles to find a partition satisfying the imbalance criteria so the refinement algorithm (Algorithm 6) performs the maximum number of refinement cycles. This large number of cycles may explain the higher quality of the final ordering.

For small α, we expect that more balanced partitions should lead to a faster parallel factorization. However, this is not evident from the factorization times shown in Figure 8.9. This may be due to the fact that the DAG-based factorization used in `HSL_MA87` is better at exploiting parallelism than traditional tree-based methods.

Based on our experiments, in `SPRAL_ND` the default setting is $\alpha = 4.0$.

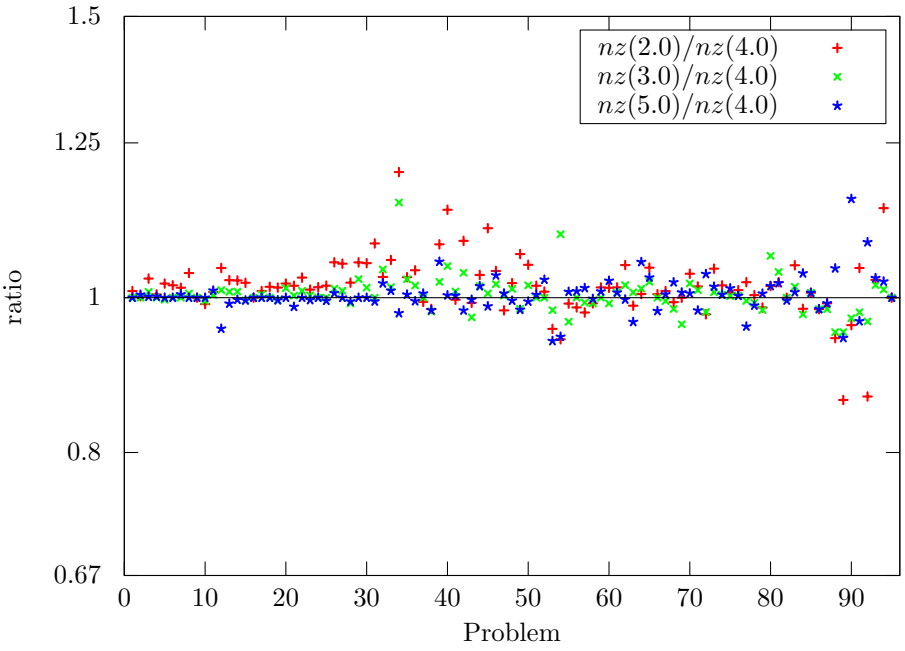

Figure 8.7: Comparison of nz for $\alpha = 2.0$, 3.0, 4.0 and 5.0 in `SPRAL_ND`. The problems are in the order given in Table A.1.

8.5 Comparison of cost functions `cost1` and `cost2`

A cost function is used to choose between candidate partitions (Section 2). In Figures 8.11 and 8.12, we compare the cost functions `cost1` (2.4) and `cost2` (2.5). Detailed results for a subset of problems that

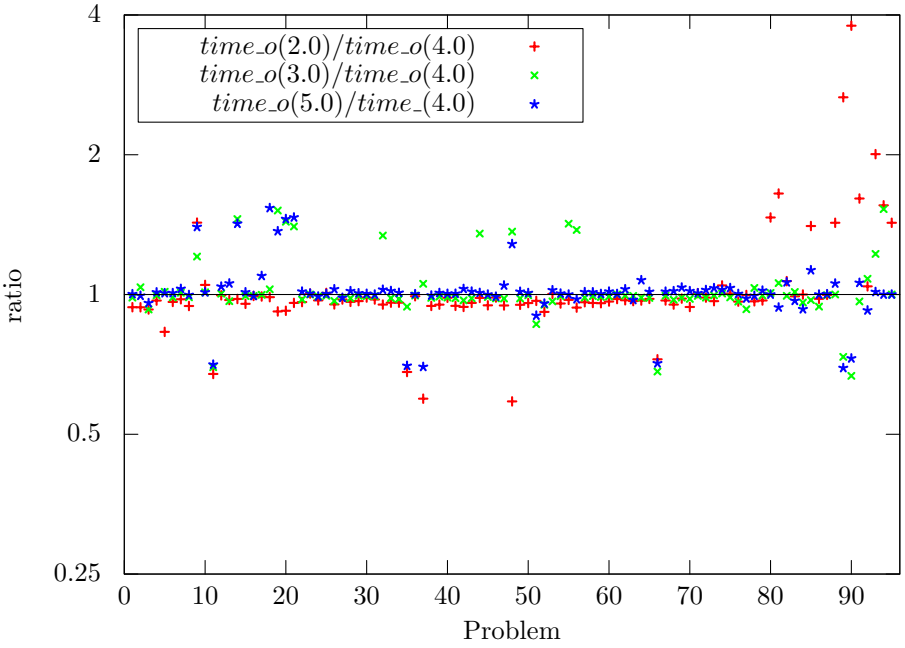

Figure 8.8: Comparison of *time_o* for $\alpha = 2.0$, 3.0, 4.0 and 5.0 in `SPRAL_ND`. The problems are in the order given in Table A.1.

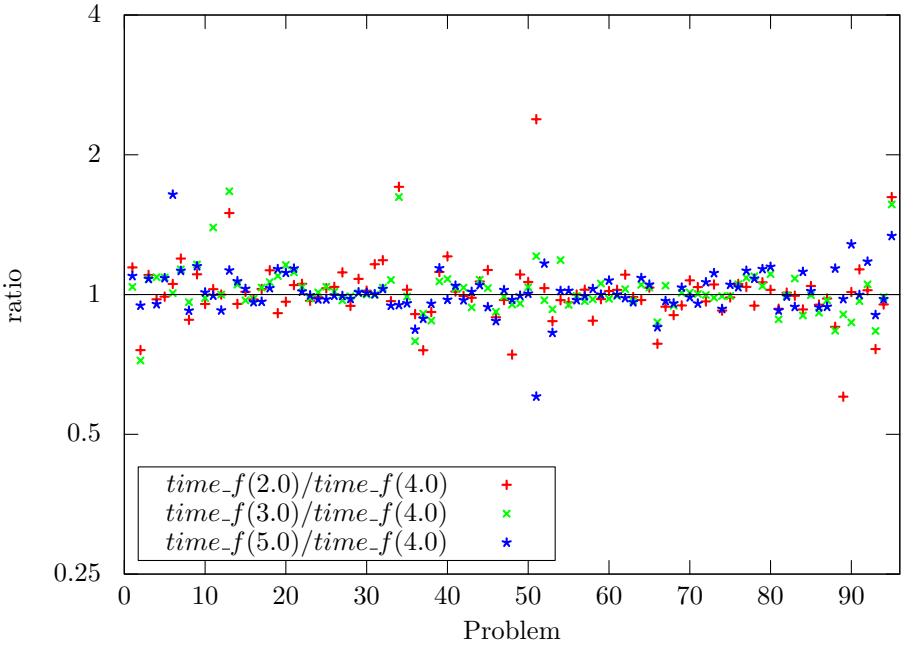

Figure 8.9: Comparison of *time_f* for $\alpha = 2.0$, 3.0, 4.0 and 5.0 in `SPRAL_ND`. The problems are in the order given in Table A.1.

Name		$\alpha = 1.0$	$\alpha = 1.5$	$\alpha = 2.0$	$\alpha = 3.0$	$\alpha = 4.0$	$\alpha = 5.0$
apache2	$nz(L)(10^6)$	212	**192**	**191**	**185**	**183**	**175**
	$nflops(10^9)$	238	220	228	209	210	**170**
	$time\_o$	11.2	3.27	**2.49**	**2.51**	**2.50**	**2.60**
	$time\_f$	1.43	**1.13**	1.32	1.32	1.32	**1.22**
	$time\_t$	13.2	4.91	**4.30**	**4.33**	**4.32**	**4.32**
Lin	$nz(L)(10^6)$	143	**107**	**105**	**103**	**102**	**102**
	$nflops(10^9)$	316	**182**	**177**	**173**	**168**	**168**
	$time\_o$	6.95	**0.85**	**0.78**	**0.79**	**0.81**	**0.83**
	$time\_f$	1.22	**0.75**	**0.77**	**0.76**	**0.73**	**0.74**
	$time\_t$	8.46	**1.80**	**1.76**	**1.75**	**1.74**	**1.76**
bone010	$nz(L)(10^6)$	1480	1700	1630	1560	**1360**	**1330**
	$nflops(10^9)$	6630	11100	10700	9790	**6320**	**5930**
	$time\_o$	58.2	**2.18**	**2.18**	**2.22**	**2.27**	**2.29**
	$time\_f$	12.7	20.5	20.3	19.3	**11.9**	**11.3**
	$time\_t$	73.3	25.1	25.0	24.0	**16.5**	**15.9**
turon_m	$nz(L)(10^6)$	**20.1**	33.6	33.2	33.6	33.3	33.4
	$nflops(10^9)$	**5.23**	21.2	21.4	21.5	21.4	21.3
	$time\_o$	3.31	**0.68**	**0.69**	**0.72**	**0.73**	**0.74**
	$time\_f$	**0.11**	0.21	0.20	0.21	0.20	0.21
	$time\_t$	3.52	**1.00**	**1.01**	**1.04**	**1.05**	**1.06**
aug3dcqp	$nz(L)(10^6)$	3.10	**2.46**	**2.41**	**2.39**	**2.36**	**2.35**
	$nflops(10^9)$	0.62	**0.37**	**0.36**	**0.35**	**0.34**	**0.34**
	$time\_o$	0.76	0.10	**0.07**	0.17	0.12	0.16
	$time\_f$	0.04	0.04	**0.04**	0.05	0.05	0.05
	$time\_t$	0.84	0.18	**0.14**	0.27	0.22	0.26
audikw_1	$nz(L)(10^6)$	**1310**	**1330**	**1360**	**1350**	**1340**	**1350**
	$nflops(10^9)$	**5880**	**6140**	6860	6610	6730	6790
	$time\_o$	28.7	**6.26**	**6.14**	**6.28**	**6.28**	**6.30**
	$time\_f$	**11.3**	**11.9**	13.0	12.6	12.7	12.7
	$time\_t$	42.9	**21.0**	**21.9**	**21.8**	**21.9**	**21.9**
crankseg_2	$nz(L)(10^6)$	**47.5**	56.0	55.8	55.5	54.9	53.9
	$nflops(10^9)$	**52.9**	77.7	77.5	75.0	74.3	71.3
	$time\_o$	1.23	**0.32**	**0.31**	**0.30**	**0.30**	**0.30**
	$time\_f$	**0.27**	0.36	0.36	0.35	0.35	0.33
	$time\_t$	1.85	**1.03**	**1.02**	**1.00**	**1.00**	**0.99**
m_t1	$nz(L)(10^6)$	**38.5**	41.4	**39.1**	**37.8**	**37.5**	**38.1**
	$nflops(10^9)$	**26.2**	32.8	27.2	**24.8**	**24.4**	**25.4**
	$time\_o$	1.20	**0.21**	**0.21**	**0.21**	**0.22**	**0.22**
	$time\_f$	**0.21**	0.23	**0.21**	**0.19**	**0.20**	0.22
	$time\_t$	1.60	**0.64**	**0.61**	**0.61**	**0.61**	**0.64**
ncvxqp7	$nz(L)(10^6)$	**28.5**	34.1	35.0	35.3	37.1	38.7
	$nflops(10^9)$	**30.1**	48.9	54.2	52.4	61.7	67.7
	$time\_o$	30.2	1.71	1.77	**1.24**	**1.24**	**1.31**
	$time\_f$	**0.27**	0.41	0.43	0.42	0.50	0.57
	$time\_t$	30.6	2.26	2.35	**1.81**	**1.90**	2.06
dictionary28	$nz(L)(10^6)$	**8.48**	**8.50**	**8.92**	**9.01**	**9.28**	10.7
	$nflops(10^9)$	**7.80**	**8.39**	9.68	9.22	10.3	14.0
	$time\_o$	10.2	2.34	1.76	**0.31**	0.46	**0.34**
	$time\_f$	**0.20**	**0.20**	0.24	**0.21**	0.24	0.31
	$time\_t$	10.5	2.64	2.12	**0.63**	0.85	0.78
lpl1	$nz(L)(10^6)$	3.03	2.99	2.89	**2.57**	**2.54**	**2.60**
	$nflops(10^9)$	0.56	0.61	0.51	**0.43**	**0.43**	**0.44**
	$time\_o$	5.25	0.48	0.37	0.36	**0.24**	**0.24**
	$time\_f$	**0.03**	0.03	**0.03**	**0.03**	**0.03**	**0.03**
	$time\_t$	5.31	0.54	0.43	0.42	**0.30**	**0.30**

Table 8.3: Results for different values of the balance parameter α. For each problem, the best statistic (and any within 10% of the best) are in bold.

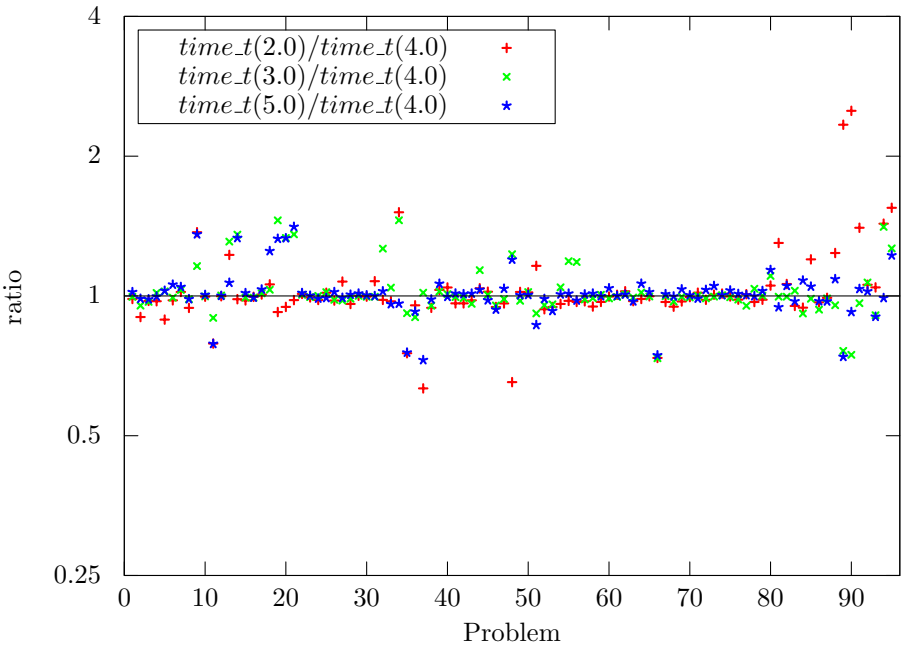

Figure 8.10: Comparison of *time_t* for $\alpha = 2.0$, 3.0, 4.0 and 5.0 in `SPRAL_ND`. The problems are in the order given in Table A.1.

illustrate a range of behaviours are given in Table 8.4. Overall, `cost1` provides a better quality ordering and is faster than `cost2`. It is thus used as the default setting.

	$nz(10^6)$		$nflops(10^9)$		*time_o*		*time_f*		*time_t*	
Name	`cost1`	`cost2`	`cost1`	`cost2`	`cost1`	`cost2`	`cost1`	`cost2`	`cost1`	`cost2`
G3_circuit	**207**	**208**	**123**	**121**	**5.61**	**6.11**	**1.31**	1.66	**7.88**	8.79
bmwcra_1	**82.6**	101	**78.5**	129	**0.39**	**0.39**	**0.44**	0.59	**1.10**	1.27
onera_dual	**11.4**	**11.7**	**3.79**	4.18	**0.64**	**0.63**	**0.09**	**0.10**	**0.78**	**0.78**
fcondp2	**72.7**	**68.8**	92.4	**75.4**	**0.33**	**0.33**	0.47	**0.42**	**1.09**	**1.04**
ct20stif	**13.3**	**13.5**	**8.03**	**8.05**	**0.20**	**0.21**	**0.11**	0.13	**0.37**	**0.41**
GaAsH6	**252**	**249**	**1650**	**1630**	**1.28**	**1.33**	**4.48**	**4.47**	**6.15**	**6.19**
gupta2	**14.1**	**13.9**	**5.60**	**5.20**	**0.47**	**0.48**	0.80	**0.62**	1.61	**1.44**
c-56	**2.48**	**2.58**	**0.42**	0.47	**0.32**	0.36	**0.04**	**0.05**	**0.41**	0.45
Andrews	**48.0**	**48.7**	**75.2**	**78.2**	**1.00**	6.59	**0.58**	**0.60**	**1.75**	7.37

Table 8.4: Comparison of the cost functions `cost1` and `cost2` for a subset of problems. For each problem, the best statistic (and any within 10% of the best) are in bold.

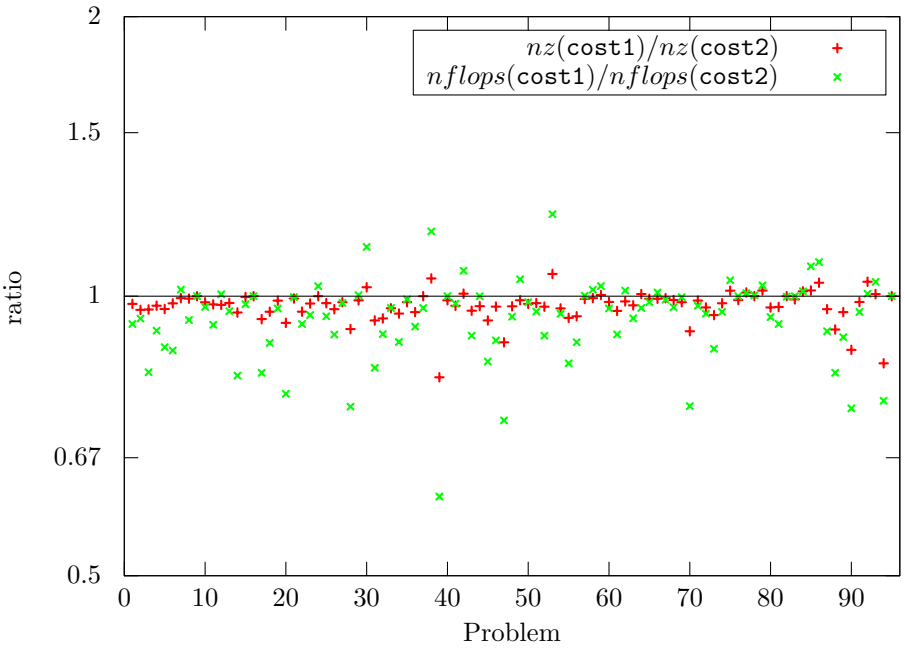

Figure 8.11: Comparison of *nz* and *nflops* for the cost functions `cost1` and `cost2`. The problems are in the order given in Table A.1.

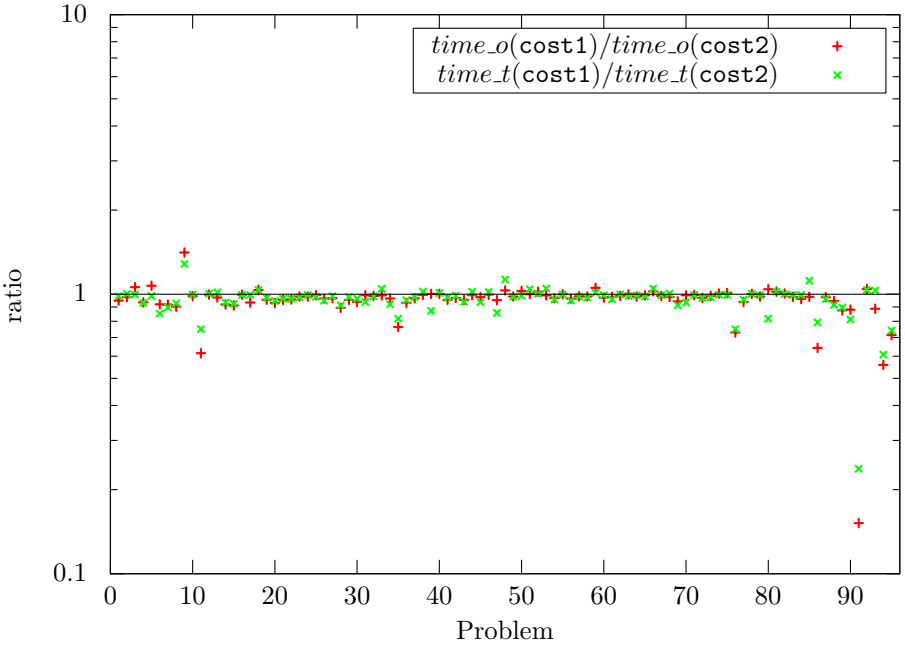

Figure 8.12: Comparison of *time_o* and *time_t* for the cost functions `cost1` and `cost2`. The problems are in the order given in Table A.1.

8.6 Comparison of SPRAL_ND with MeTiS

One of the aims of this project was to develop a nested dissection package that could provide an open source alternative to the well known and widely used MeTiS nested dissection package [25, 27]. In this section, comparisons are made between the most recent version of MeTiS (Version 5.1.0) and SPRAL_ND using the non-multilevel approach (in Appendix B, we include comparisons with SPRAL_ND using the multilevel approach, see Figures B.14-B.17). In terms of ordering times (Figure 8.14), SPRAL_ND is faster than MeTiS but the factors are denser, particularly for problems for which %*band* is not small. This is reflected in the factorization times (Figure 8.15) and the total times (Figure 8.16), with the faster (serial) ordering times of SPRAL_ND using the non-multilevel approach dominating the slower (parallel) factorization times to give a faster total time for SPRAL_ND for many of the test examples.

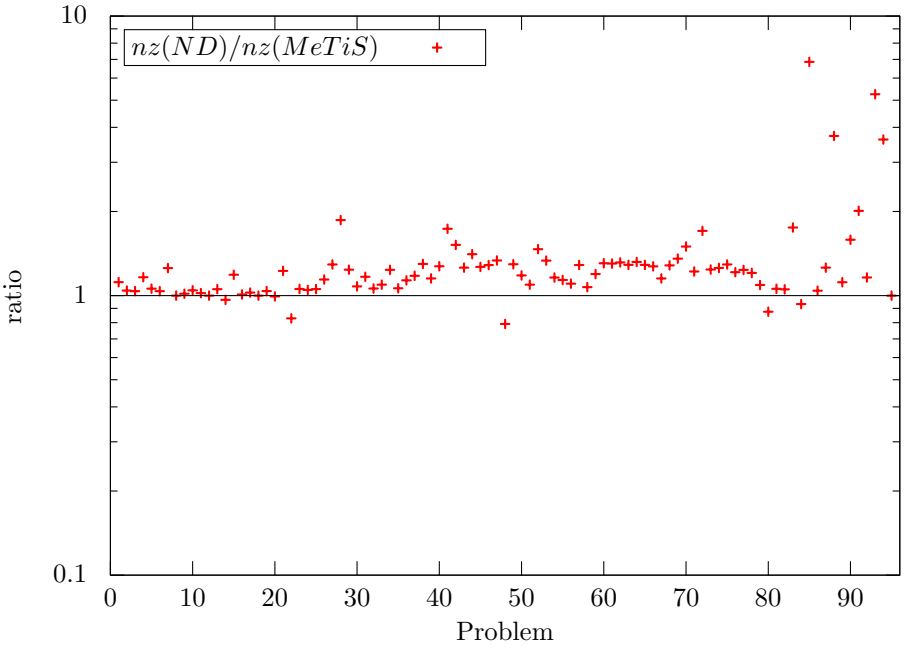

Figure 8.13: Comparison of nz for SPRAL_ND (denoted by ND) using the non-multilevel approach with MeTiS. The problems are in the order given in Table A.1.

Finally, we give some timings comparisons in Figures 8.17-8.19 for a typical desktop machine. Specifically, we run on a machine with a single i7-4790 quad core processor and 16 GB of memory. We employ the gfortran (Version 4.9.3) complier with options -g -O2 -fopenmp and MKL BLAS (Version 11.3.047). We again see that the SPRAL_ND ordering time (with the non-multilevel approach) is less than for MeTiS. However, the higher quality MeTiS ordering now results in much greater savings in the factorization times. Consequently, for problems with small %*band*, the total time for SPRAL_ND is still less than for MeTiS but for larger %*band* there is no clear winner.

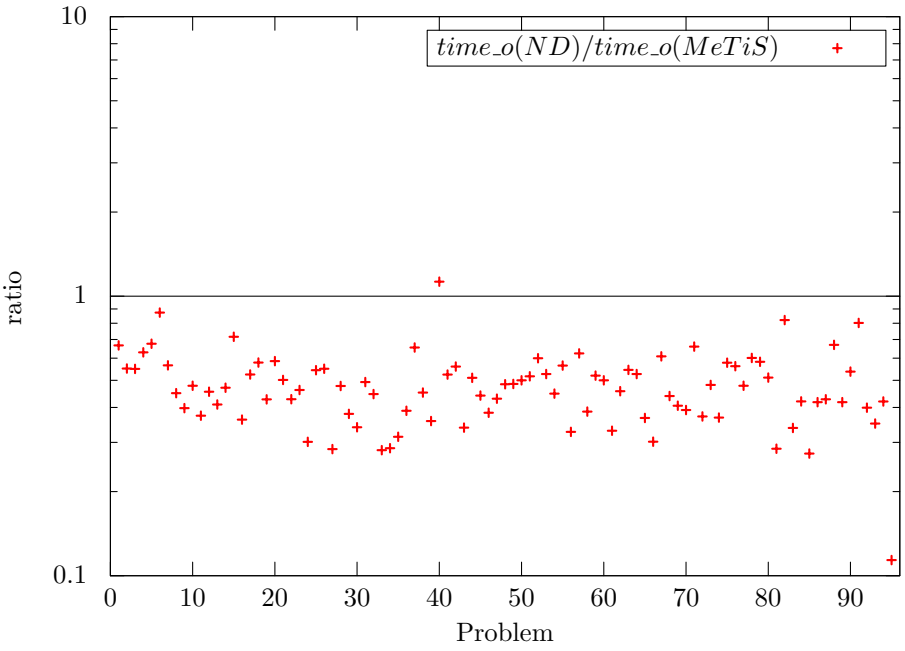

Figure 8.14: Comparison of *time_o* for `SPRAL_ND` (denoted by ND) using the non-multilevel approach with `MeTiS`. The problems are in the order given in Table A.1.

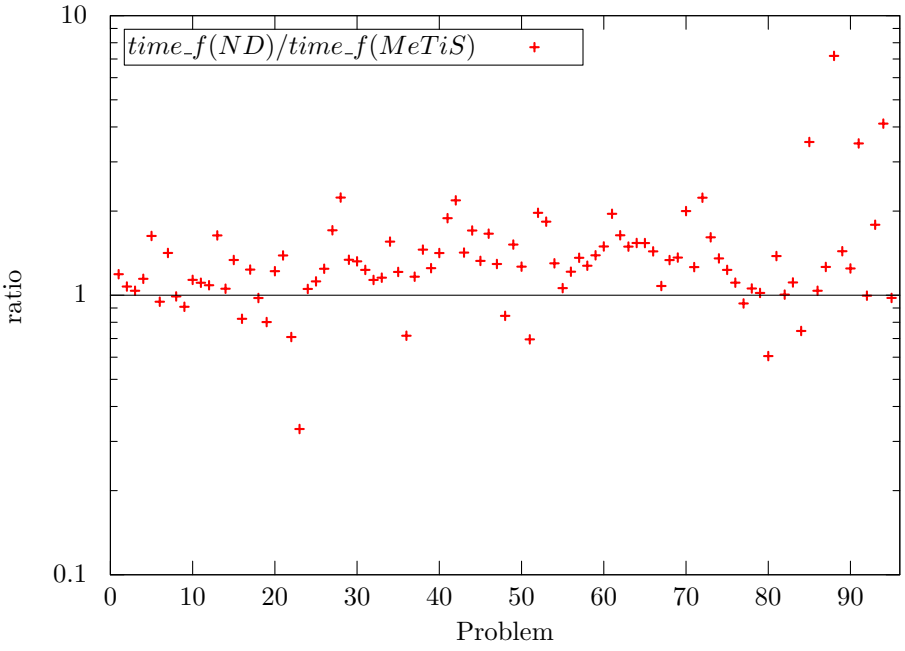

Figure 8.15: Comparison of *time_f* for `SPRAL_ND` (denoted by ND) using the non-multilevel approach with `MeTiS`. The problems are in the order given in Table A.1.

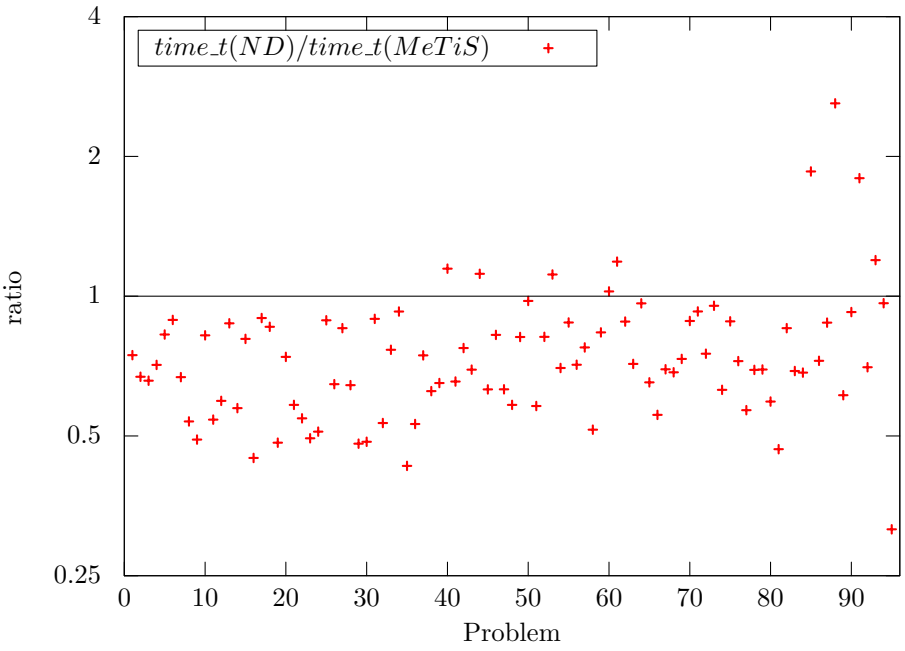

Figure 8.16: Comparison of *time_t* for `SPRAL_ND` (denoted by ND) using the non-multilevel approach with `MeTiS`. The problems are in the order given in Table A.1.

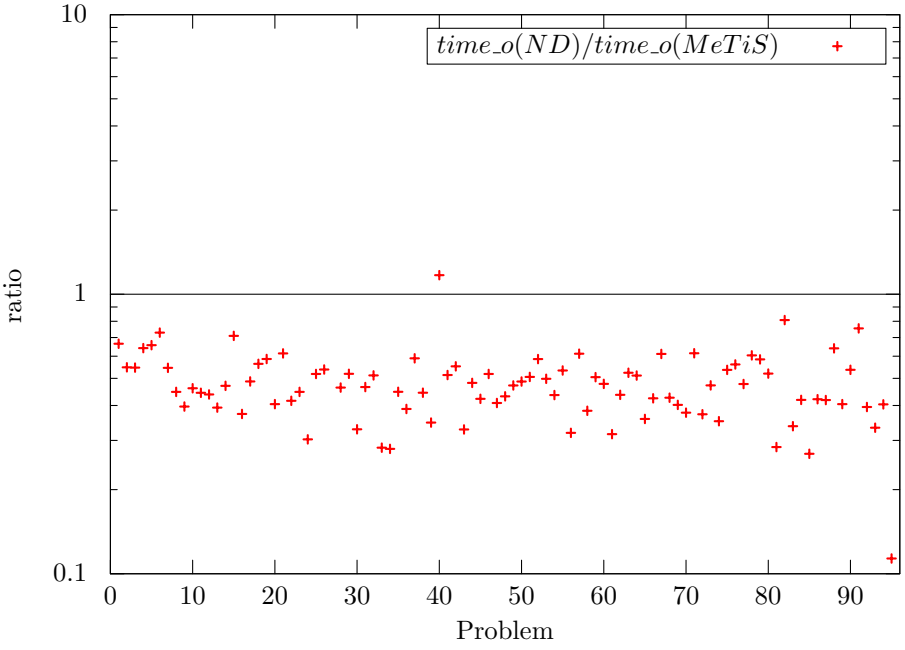

Figure 8.17: Comparison of *time_o* for `SPRAL_ND` (denoted by ND) using the non-multilevel approach with `MeTiS` on a desktop machine. The problems are in the order given in Table A.1.

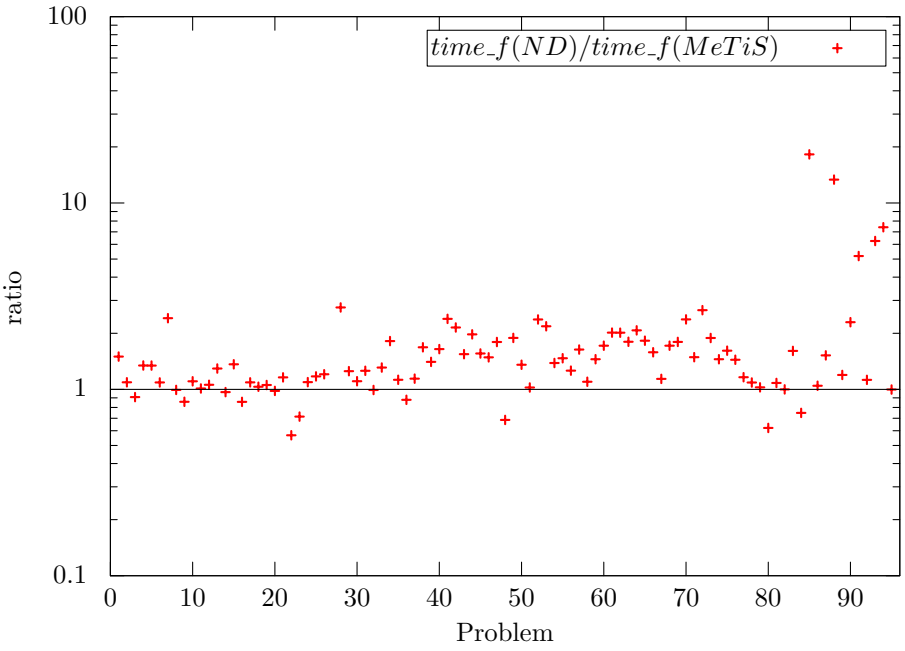

Figure 8.18: Comparison of *time_f* for `SPRAL_ND` (denoted by ND) using the non-multilevel approach with `MeTiS` on a desktop machine. The problems are in the order given in Table A.1.

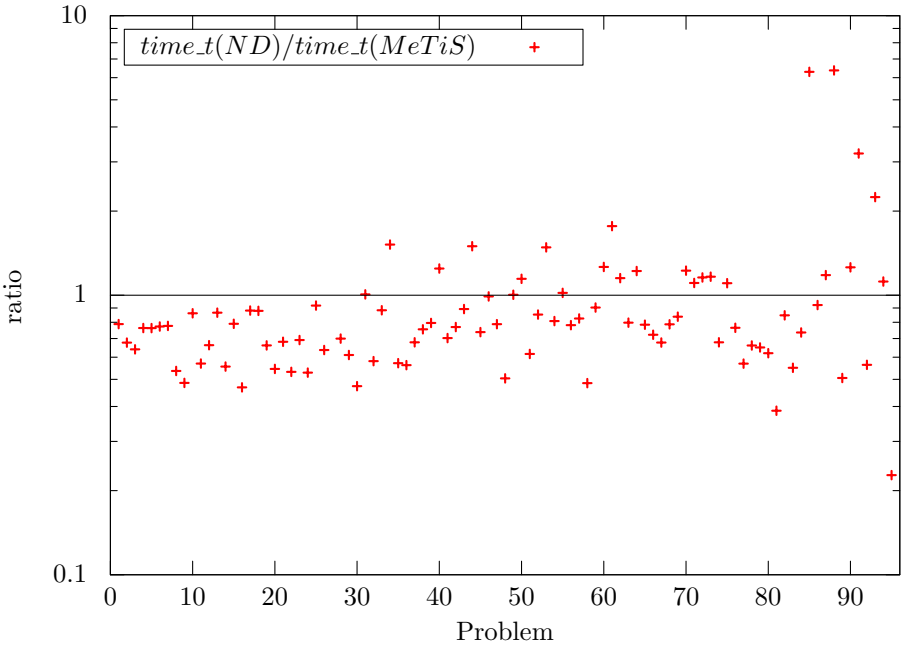

Figure 8.19: Comparison of *time_t* for `SPRAL_ND` (denoted by ND) using the non-multilevel approach with `MeTiS` on a desktop machine. The problems are in the order given in Table A.1.

9 Concluding remarks

This report has described in detail the development of a new nested dissection package `SPRAL_ND`. We have explored a number of ideas, including the use of half-level sets, the avoidance of multilevels, and different refinement techniques. It is clear from our discussions that a modern nested dissection algorithm is built using a large number of other algorithms (many of which are used in other contexts) and that parameters are needed to control how and when these building blocks are applied. A balance has to be struck between the quality of the computed ordering and the time taken to compute it. In different applications the balance between these two requirements may be very different (most notably if the ordering is to be used for a single factorization or for many factorizations). In particular, we have shown that, unless the problem has a small bandwidth, using a non-multilevel approach generally produces a poorer quality ordering than the multilevel approach. However, the former is significantly faster to compute, resulting in the total solution time for a single (parallel) factorization and solve being less for the non-multilevel ordering than for multilevel ordering.

Using a set of practical problems with a range of characteristics we have found that the quality of an ordering for a given problem can be very sensitive to the parameter choices. This makes it difficult to provide default values that work well for all problems, but we have presented results that show how the ordering time and quality can be affected by the choices.

Code Availability

A development version of the nested dissection ordering software (with user documentaion) that is used in this paper may be checked out of our source code repository using the following command:

```
svn co -r640 http://ccpforge.cse.rl.ac.uk/svn/spral/branches/nested_dissection
```

This code has not been optimised for high performance and we do not currently plan to release it as part of the HSL or SPRAL libraries that we develop and maintain at the Rutherford Appleton Laboratory (see `http://www.hsl.rl.ac.uk/` and `http://www.numerical.rl.ac.uk/spral/`).

References

[1] P. Amestoy, H. Dollar, J. Reid, and J. Scott, *An approximate minimum degree algorithm for matrices with dense rows*, Technical Report RAL-TR-2007-020, Rutherford Appleton Laboratory, 2007.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff, *An approximate minimum degree ordering algorithm*, SIAM J. Matrix Anal. Appl., 17 (1996), pp. 886–905.

[3] C. Ashcraft, *Compressed graphs and the minimum degree algorithm*, SIAM J. Sci. Stat. Comput., 16 (1995), pp. 1401–1411.

[4] C. Ashcraft and J. W. H. Liu, *A partition improvement algorithm for generalized nested dissection*, Technical Report BCSTECH-94-020, Boeing Computer Services, Seattle, 1994.

[5] C. Ashcraft and J. W. H. Liu, *Applications of the Dulmage-Mendelsohn decomposition and network flow to graph bisection improvement*, SIAM J. Matrix Anal. Applic., 19 (1998), pp. 325–354.

[6] S. T. Barnard, A. Pothen, and H. D. Simon, *A spectral algorithm for envelope reduction of sparse matrices*, Numer. Lin. Alg. Appl., 2 (1995), pp. 317–334.

[7] S. T. Barnard and H. D. Simon, *A fast multilevel implmentation of recursive spectral bisection for partitioning unstructured problems*, Concurrency: Practice and Experience, 6 (1994), pp. 101–117.

[8] J. L. Carmen, *An AMD preprocessor for matrices with some dense rows and columns.* `http://www.netlib.org/linalg/amd/amdpre.ps`.

[9] C. Chevalier and F. Pellegrini, *PT-SCOTCH: a tool for efficient parallel graph ordering*, Parallel Computing, 34 (2008), pp. 318–331.

[10] E. Cuthill and J. McKee, *Reducing the bandwidth of sparse symmetric matrices*, in Proc. 24th Nat. Conf., ACM Publications, 1969, pp. 157–172.

[11] T. A. Davis, *Direct Methods for Sparse Linear Systems*, vol. 2 of Fundamentals of Algorithms, SIAM, Philadelphia, PA, 2006.

[12] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Trans. Math. Softw., 38 (2011), pp. 1–25.

[13] H. S. Dollar and J. A. Scott, *A note on fast approximate minimum degree orderings for matrices with some dense rows*, Numer. Lin. Alg. Appl., 17 (2010), pp. 43–55.

[14] I. S. Duff and J. A. Scott, *Towards an automatic ordering for a symmetric sparse direct solver*, Technical Report RAL-TR-2006-001, Rutherford Appleton Laboratory, 2006.

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

[16] A. George, *Nested dissection of a regular finite-element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.

[17] A. George and J. Liu, *Computer solution of large sparse positive definite systems*, Prentice-Hall, Englewood Cliffs, N.J., 1981.

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

[19] B. Hendrickson and R. Leland, *A multilevel algorithm for partitioning graphs*, Technical Report SAND93-1301, Sandia National Laboratories, Allbuquerque, NM, 1993.

[20] B. Hendrickson and E. Rothberg, *Improving the runtime and quality of nested dissection ordering*, SIAM J. Sci. Comput., 20 (1998), pp. 468–489.

[21] J. D. Hogg, J. K. Reid, and J. A. Scott, *Design of a multicore sparse cholesky factorization using DAGs*, SIAM J. Sci. Comput., 32 (2010), pp. 3627–3649.

[22] J. D. Hogg and J. A. Scott, *A modern analyse phase for sparse tree-based direct methods*, Numer. Lin. Alg. Appl., 20 (2013), pp. 397–412. published online March 2012.

[23] HSL, *A collection of Fortran codes for large-scale scientific computation.* `http://www.hsl.rl.ac.uk/`, 2013.

[24] Y. F. Hu and J. A. Scott, *A multilevel algorithm for wavefront reduction*, SIAM J. Sci. Comput., 23 (2001), pp. 1352–1375.

[25] G. Karypis and V. Kumar, *METIS: Unstructured graph partitioning and sparse matrix ordering system*, Technical Report TR 95-035, University of Minnesota, 1995.

[26] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM J. Sci. Comput., (1998), pp. 359–392.

[27] G. Karypis and V. Kumar, *METIS: A software package for partitioning unstructured graphs, partitioning meshes and computing fill-reducing orderings of sparse matrices - version 4.0*, 1998.

[28] J. L. R. Ford and D. R. Fulkerson, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[29] J. W. H. Liu, *Modification of the minimum-degree algorithm by multiple elimination*, ACM Trans. Math. Softw., 11 (1985), pp. 141–153.

[30] H. M. Markowitz, *The elimination form of the inverse and its application to linear programming*, Management Science, 3 (1957), pp. 255–269.

[31] S. B. Patkar, V. Kumar, H. Kaur, and B. Hore, *A graph partitioning system for natural unbalanced partitions*, in Proceedings of the 2002 WSEAS Int. Conf. on Electronics, Control and Signal Processing, 2002.

[32] F. Pellegrini, *SCOTCH 6.0*, 2012. `http://www.labri.fr/perso/pelegrin/scotch/`.

[33] A. Pirova and I. Meyerov, *MORSy – a new tool for sparse matrix reordering*, in OPT-i An International Conference on Engineering and Applied Sciences Optimization, M. Papadrakakis, M. G. Karlaftis, and N. D. Lagaros, eds., 2014, pp. 1952–1963.

[34] J. K. Reid and J. A. Scott, *Ordering symmetric sparse matrices for small profile and wavefront*, Int. J. Numer. Meth. Engng, 45 (1999), pp. 1737–1755.

[35] J. W. Ruge and K. Stüben, *Algebraic multigrid (AMG)*, in Multigrid Methods, S. F. McCormick, ed., vol. 3 of Frontiers in Applied Mathematics, SIAM, Philadelphia, PA, 1987, pp. 73–130.

[36] W. Tinney and J. Walker, *Direct solutions of sparse network equations by optimally ordered triangular factorization*, Proc. IEEE, 55 (1967), pp. 1801–1809.

[37] M. Yannakakis, *Computing the minimum fill-in is NP-complete*, SIAM J. Alg. Discrete Meth., 2 (1981), pp. 77–79.

A Test problems

In Table A.1 we list our test problems along with their characteristics. The problems are from the University of Florida Sparse Matrix Collection and are chosen to represent a wide range of sparsity structures. Given a test matrix, we immediately perform the update $A \leftarrow A + A^T$ to form a symmetrized version. $\widetilde{A}$ is the resulting matrix after the dense rows have been removed (Section 3.1), the matrix has been compressed (Section 3.2) and diagonal entries removed. We provide the following information about each test problem:

- Problem identifier
- n : the order of the problem as supplied by [12]
- ne : the number of nonzero entries in the symmetrized version of the problem
- n_d : the number of dense rows removed
- $\widetilde{n}$: the order of $\widetilde{A}$
- $\widetilde{ne}$: the number of nonzero entries in $\widetilde{A}$
- *comps* : the number of independent components in $\widetilde{A}$
- n_{maxc} : the order of the largest independent component in $\widetilde{A}$
- ne_{maxc} : the number of nonzero entries in the largest independent component in $\widetilde{A}$
- $\%band$: the bandwidth as a percentage of n_{maxc} of the largest independent component in $\widetilde{A}$ after the Reverse Cuthill-McKee algorithm [10] has been applied [17].

The test problems are ordered such that $\%band$ is increasing.

Table A.1: Test problems.

Identifier	n	ne	n_d	$\widetilde{n}$	$\widetilde{ne}$	*comps*	n_{maxc}	ne_{maxc}	$\%band$
thermal2	4904179	4904179	0	1228045	7352268	959	1227087	7352268	0.07
parabolic_fem	2100225	2100225	0	525825	3148800	1	525825	3148800	0.10
ecology1	2998000	2998000	0	1000000	3996000	1	1000000	3996000	0.10
tmt_sym	2903837	2903837	0	726713	4354248	1	726713	4354248	0.16
helm2d03	1567096	1567096	0	392257	2349678	1	392257	2349678	0.25
darcy003	1167685	1167685	0	389775	1866520	1	389775	1866520	0.30
G3_circuit	4623152	4623152	0	1585478	6075348	1	1585478	6075348	0.32
cont-300	539396	539396	0	180895	897598	1	180895	897598	0.33
case39	526139	526139	20	40165	610818	2	40164	610818	0.36
BenElechi1	6698185	6698185	0	40985	324330	7	40979	324330	0.38
shallow_water1	204800	204800	0	81920	245760	1	81920	245760	0.39
apache2	2766523	2766523	0	715176	4102694	1	715176	4102694	0.41
af_shell5	9046865	9046865	0	100971	602584	1	100971	602584	0.49
cont-201	239596	239596	0	80595	398398	1	80595	398398	0.50
rail_79841	316881	316881	0	79841	474080	1	79841	474080	0.52
TSOPF_FS_b39_c30	1575639	1575639	20	120165	1829818	2	120164	1829818	0.54
s4dkt3m2	1921955	1921955	0	15251	90500	1	15251	90500	0.66
srb1	1508538	1508538	0	9154	73128	1	9154	73128	0.96
pwt	181313	181313	0	36515	289548	57	36459	289548	0.97
apache1	311492	311492	0	80800	461384	1	80800	461384	0.98
ford2	322442	322442	0	97906	420858	1	97906	420858	0.98
Lin	1011200	1011200	0	256000	1510400	1	256000	1510400	1.09
pwtk	5926171	5926171	0	41531	442260	1	41531	442260	1.17
nasasrb	1366097	1366097	0	24954	551626	1	24954	551626	1.15
s3dkq4m2	2455670	2455670	0	15251	120500	1	15251	120500	1.28
G2_circuit	438388	438388	0	150102	576572	1	150102	576572	1.30
Flan_1565	59485419	59485419	0	521598	12523518	1	521598	12523518	1.31

Name	n	ne	n_d	$\widetilde{n}$	$\widetilde{ne}$	$comps$	n_{maxc}	ne_{maxc}	$\%band$
d_pretok	885416	885416	0	182730	1512512	1	182730	1512512	1.48
stokes128	295938	295938	0	49666	525312	1	49666	525312	1.53
Dubcova3	1891669	1891669	0	65025	965200	1	65025	965200	1.54
halfb	6306219	6306219	0	38556	321406	1	38556	321406	1.55
finan512	335872	335872	0	74752	522240	1	74752	522240	1.62
ibm_matrix_2	558004	558004	2	17150	99560	1	17150	99560	1.75
bone010	36326514	36326514	0	327862	7616186	2	327856	7616166	1.77
struct3	613632	613632	0	41644	681086	2	25153	475830	1.81
FEM_3D_thermal2	1818600	1818600	0	147900	3341400	1	147900	3341400	1.89
rajat10	80202	80202	0	30202	100202	1	30202	100202	1.93
cfd2	1605669	1605669	0	123440	2964458	1	123440	2964458	2.02
bmwcra_1	5396386	5396386	0	53877	1411816	1	53877	1411816	2.08
msdoor	10328399	10328399	0	60939	375914	1	60939	375914	2.24
turon_m	912345	912345	0	189924	1557062	1	189924	1557062	2.28
tandem_dual	277281	277281	0	94069	366424	1	94069	366424	2.32
xenon2	2012076	2012076	0	64170	519624	55	64116	519624	2.37
fullb	5953632	5953632	0	33442	294498	1	33442	294498	2.39
gas_sensor	885141	885141	0	66889	1635574	1	66889	1635574	2.65
oilpan	1835470	1835470	0	10536	62876	1	10536	62876	2.70
qa8fk	863353	863353	0	66113	1594182	1	66113	1594182	2.72
aug3dcqp	77829	77829	0	35543	100572	2	35536	100560	2.80
bmw3_2	5757996	5757996	0	55473	708118	1	55473	708118	2.93
ship_003	4103881	4103881	0	20275	204208	1	20275	204208	3.04
filter3D	1406808	1406808	0	106224	2592456	1	106224	2592456	3.16
onera_dual	252384	252384	0	85567	333634	1	85567	333634	3.17
fcondp2	5748069	5748069	0	33913	282912	1	33913	282912	3.66
water_tank	1078497	1078497	0	46193	1282318	1	46193	1282318	3.83
pkustk10	2194830	2194830	0	13446	106248	1	13446	106248	3.87
3dtube	1629474	1629474	12	15895	356052	2	15880	355956	3.97
copter2	407714	407714	0	55476	704476	1	55476	704476	3.97
dawson5	531157	531157	0	46014	824490	307	20133	420288	3.98
bmw7st_1	3740507	3740507	0	30145	339682	2	30144	339682	3.98
shipsec8	3384159	3384159	0	19532	171428	1	19532	171428	4.02
audikw_1	39297771	39297771	0	314335	8303792	1	314335	8303792	4.06
troll	6099282	6099282	0	48435	660224	1	48435	660224	4.08
cfd1	949510	949510	0	70656	1757708	1	70656	1757708	4.08
x104	5138004	5138004	0	17260	246950	1	17260	246950	4.16
gearbox	4617075	4617075	0	56175	1386284	6	43427	1179076	4.42
bcsstk32	1029655	1029655	0	14821	226974	1	14821	226974	4.84
ckt11752_dc_1	193659	193659	7	49591	260766	142	49363	260592	5.76
engine	2424822	2424822	0	47857	475040	1	47857	475040	6.12
ct20stif	1375396	1375396	0	17723	356666	1	17723	356666	6.32
boneS01	3421188	3421188	0	39672	644420	1	39672	644420	6.32
crankseg_2	7106348	7106348	16	11996	539402	1	11996	539402	6.69
t3dh_a	2215638	2215638	0	74614	3890844	1	74614	3890844	6.72
m_t1	4925574	4925574	0	17044	299194	1	17044	299194	6.76
F2	2682895	2682895	0	27433	829668	1	27433	829668	6.92
crankseg_1	5333507	5333507	4	10929	478612	1	10929	478612	6.99
helm3d01	230335	230335	0	32226	396218	1	32226	396218	7.61
hcircuit	309362	309362	3	80267	260234	1458	71087	239652	7.61
rajat23	334325	334325	35	107342	384190	658	106250	383096	7.93
rajat22	118951	118951	10	38058	132480	1027	36429	130992	7.89
CO	3943588	3943588	0	221119	7444938	1	221119	7444938	8.54
net4-1	1265035	1265035	30	88191	2089940	136	5386	140010	8.87
lung2	383106	383106	0	54730	109458	1	54730	109458	9.44
trans5	488953	488953	3	116832	327130	41624	74367	324512	13.64
GaAsH6	1721579	1721579	713	60636	2340144	1	60636	2340144	13.97
gupta2	2155175	2155175	824	60443	485774	3156	48185	467568	16.69
nd12k	7128473	7128473	0	36000	14184946	1	36000	14184946	18.24
pkustk14	7494215	7494215	0	34134	865252	1	34134	865252	20.60
ncvxqp7	312481	312481	0	87500	524962	1	87500	524962	20.61
c-56	208405	208405	42	35631	287814	239	35393	287814	21.01

Name	n	ne	n_d	$\widetilde{n}$	$\widetilde{ne}$	$comps$	n_{maxc}	ne_{maxc}	$\%band$
dictionary28	89038	89038	0	48465	166362	17903	24557	140410	22.03
Andrews	410077	410077	0	59999	700136	1	59999	700136	24.93
c-73	724348	724348	22	168968	765914	433	168536	765914	27.32
gupta1	1098006	1098006	433	30814	219200	3035	23320	210280	30.52
lpl1	180248	180248	0	32415	295072	3	32234	294016	41.75
ins2	1530448	1530448	12	309400	1093950	5525	56	198	94.64

B Performance Profiles and Additional Results

The performance ratio for an algorithm on a particular problem is the performance measure for that algorithm divided by the smallest performance measure for the same problem over all the algorithms being tested (here we are assuming that the performance measure is one for which smaller is better, for example, the flop count or time taken). The performance profile is the set of functions $\{p_i(f) : f \in [1, \infty)\}$, where $p_i(f)$ is the proportion of problems where the performance ratio of the ith algorithm is at most f. Thus $p_i(f)$ is a monotonically increasing function taking values in the interval $[0, 1]$. If we are just interested in the number of wins, we need only compare the values of $p_i(1)$ for all the algorithms but, if we are interested in algorithms with a high probability of success, we should choose the ones for which p_i^* has the largest values. In our performance profile plots, we use a logarithmic scale in order to observe the performance of the algorithms over a large range of f while still being able to discern in some detail what happens for small f.

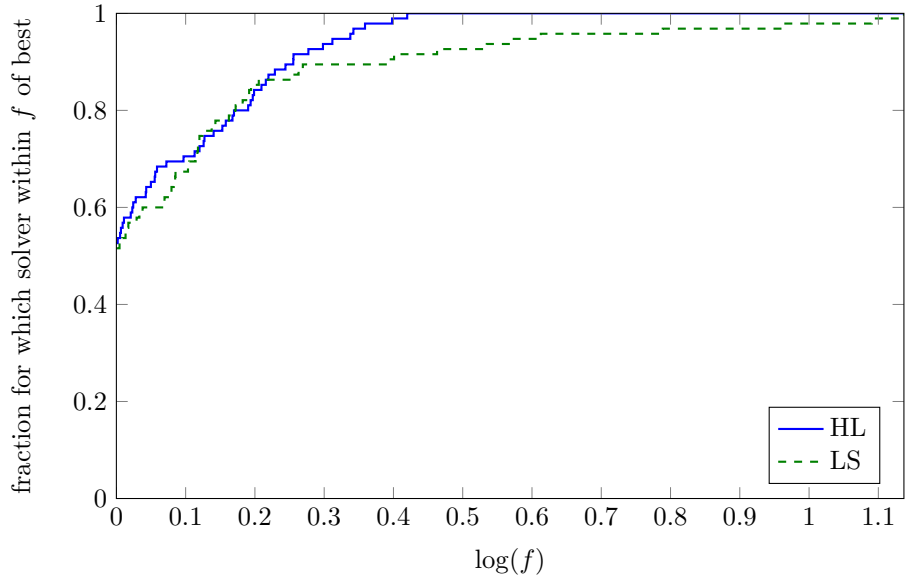

Figure B.1: *nflops* for coarse partitioning method half-level set (HL) vs level set (LS)

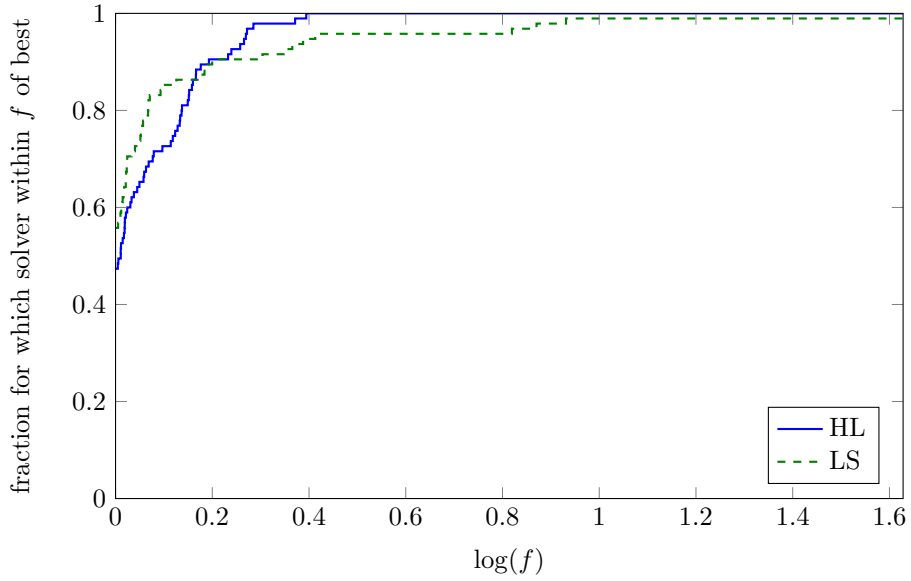

Figure B.2: *time_o* for coarse partitioning method half-level set (HL) vs level set (LS)

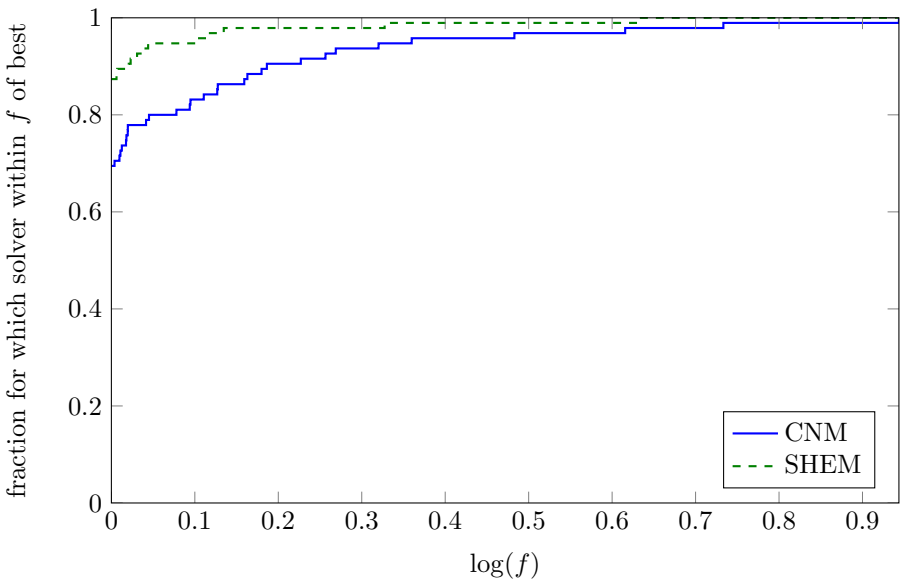

Figure B.3: *nflops* for matching method common neighbour matching (CNM) vs sorted heavy-edge matching (SHEM)

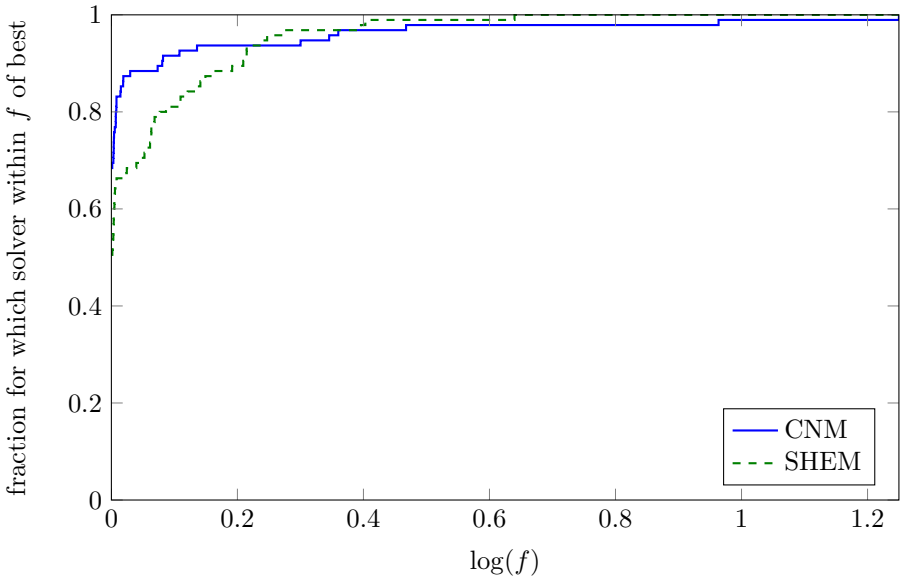

Figure B.4: *time_o* for matching method common neighbour matching (CNM) vs sorted heavy-edge matching (SHEM)

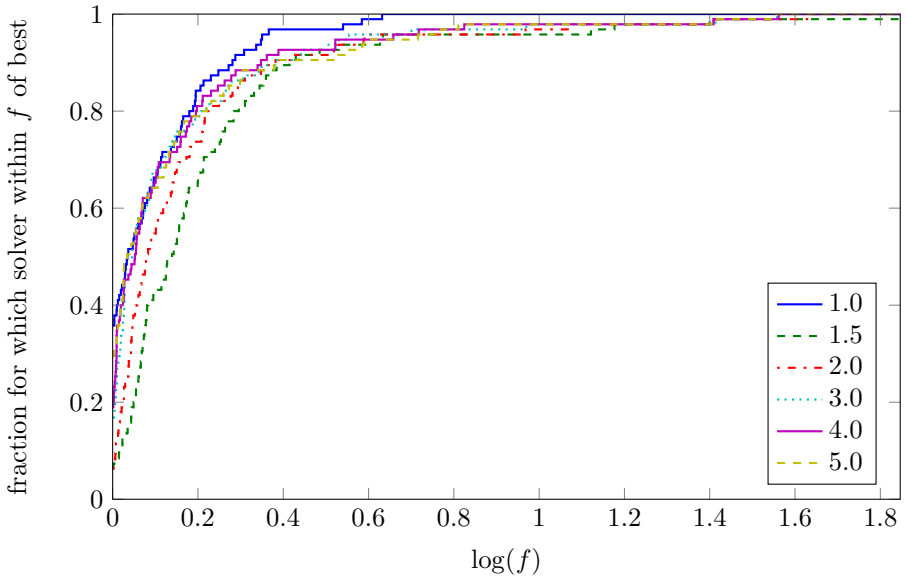

Figure B.5: *nflops* for varying values of balance parameter α

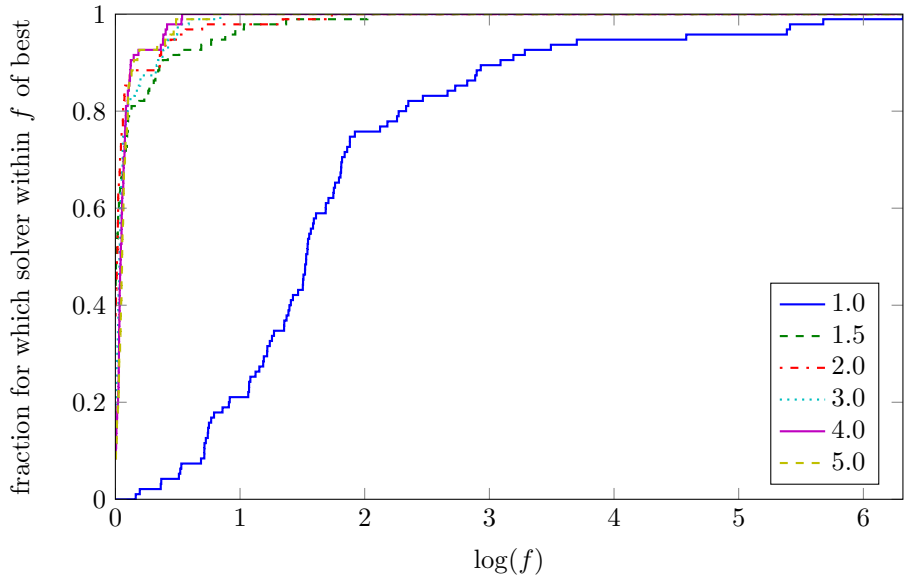

Figure B.6: *time_o* for varying values of balance parameter α

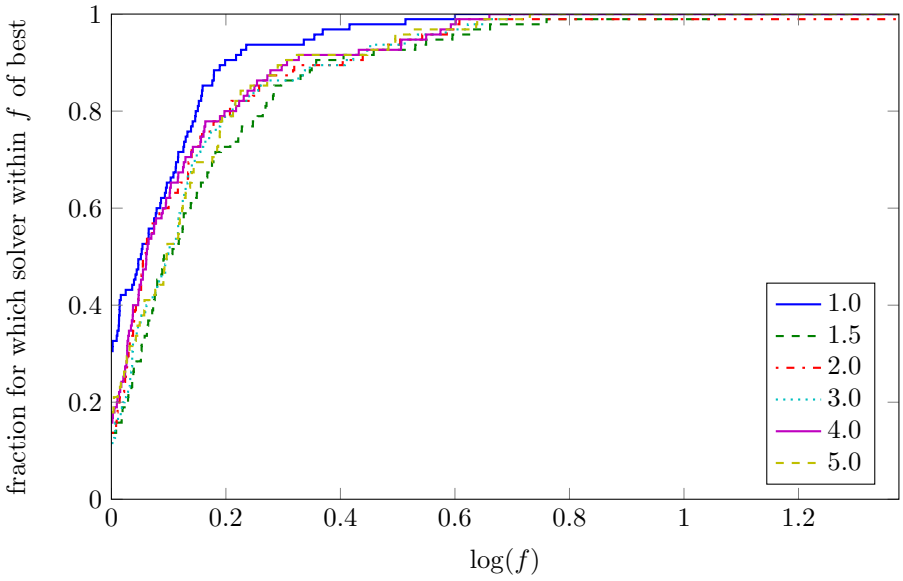

Figure B.7: *time_f* for varying values of balance parameter α

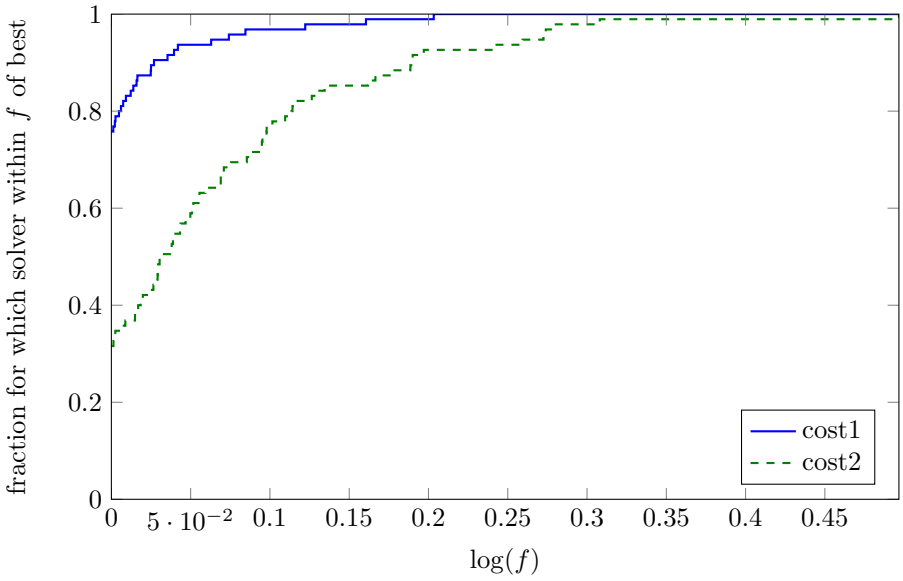

Figure B.8: *nflops* for different cost functions

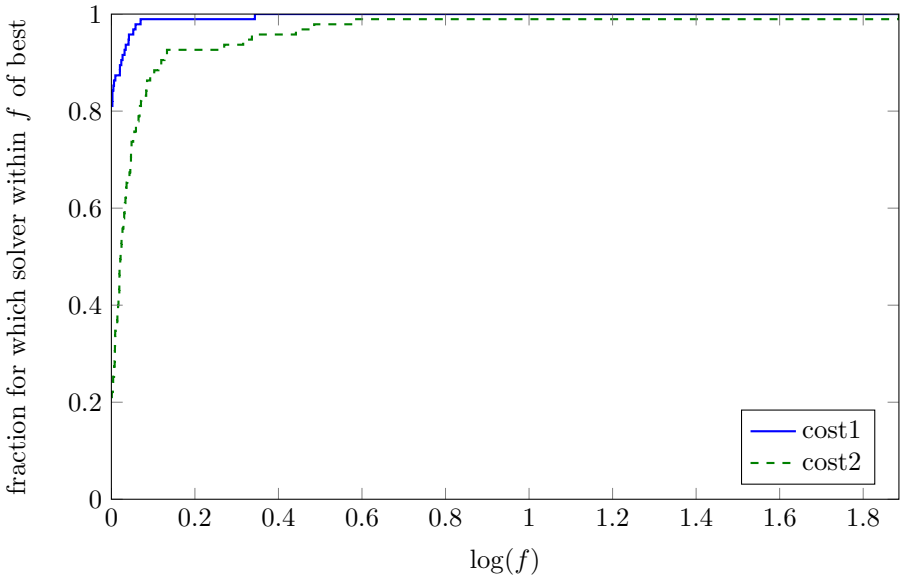

Figure B.9: *time_o* for different cost functions

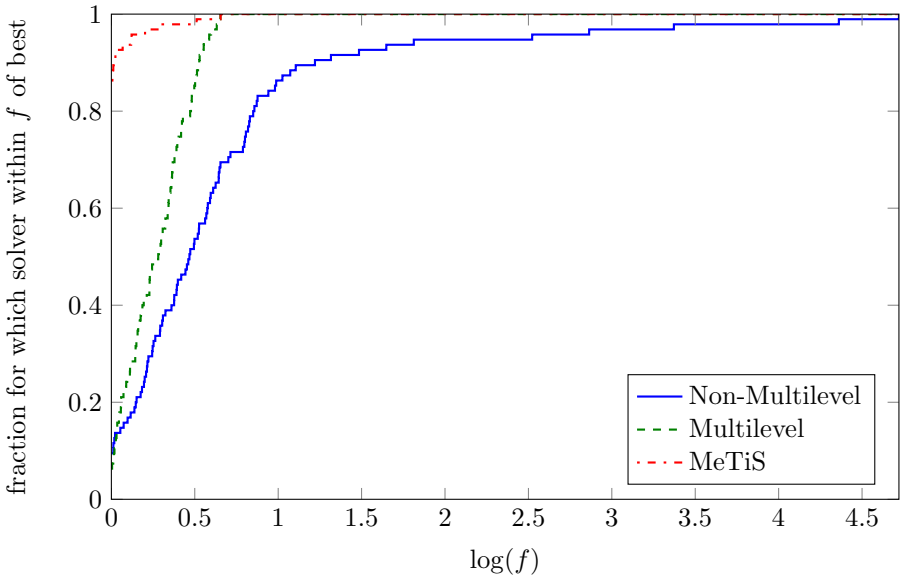

Figure B.10: *nflops* comparing `SPRAL_ND` with the non-multilevel and multilevel approaches and `MeTiS`

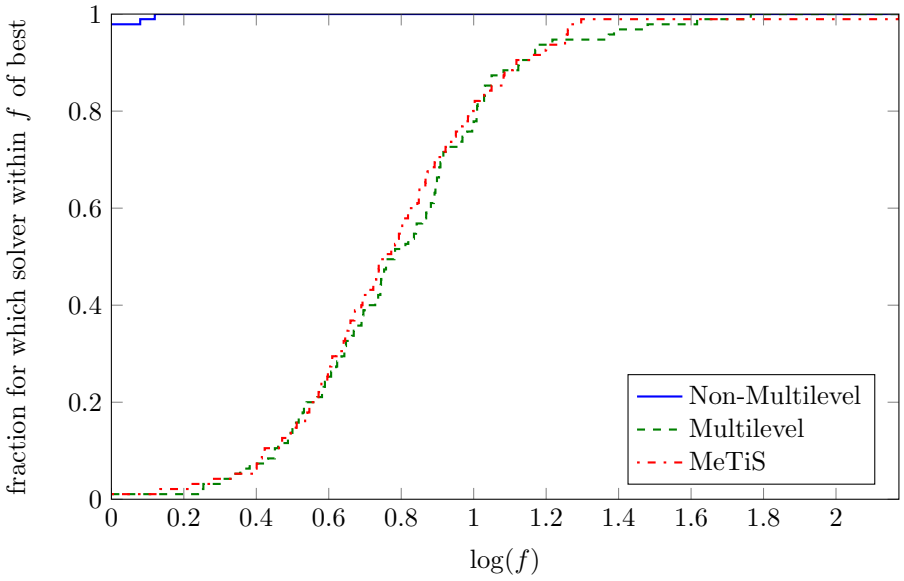

Figure B.11: *time_o* comparing `SPRAL_ND` with the non-multilevel and multilevel approaches and `MeTiS`

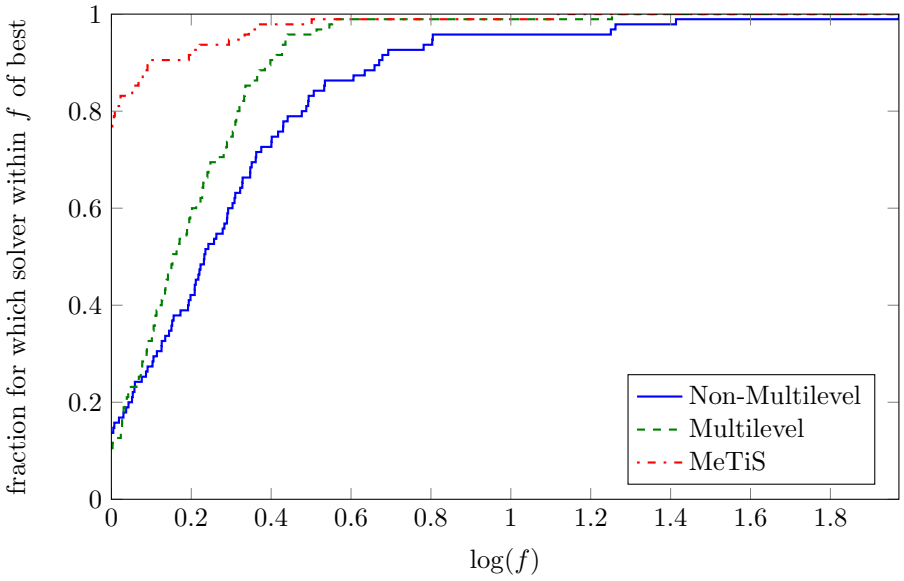

Figure B.12: *time_f* comparing `SPRAL_ND` with the non-multilevel and multilevel approaches and `MeTiS`

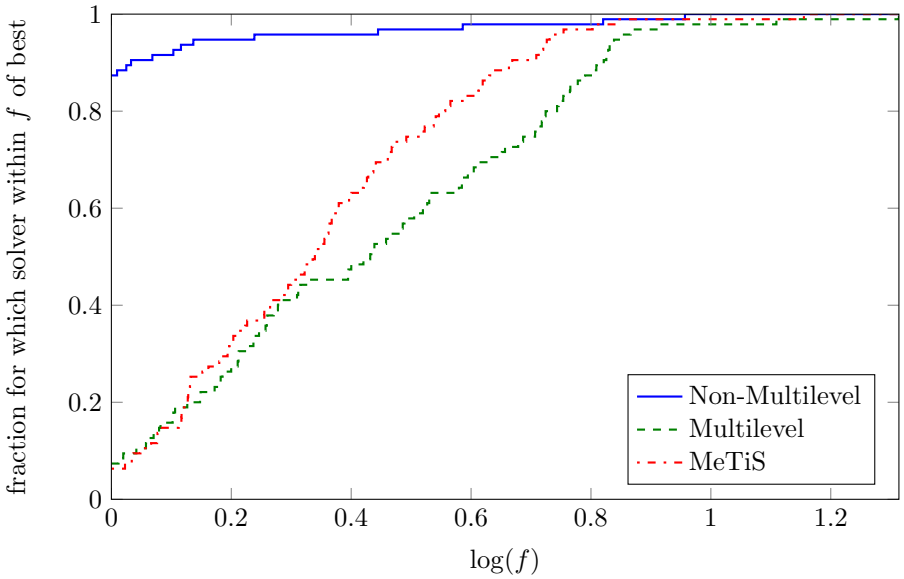

Figure B.13: *time_t* comparing `SPRAL_ND` with the non-multilevel and multilevel approaches and `MeTiS`

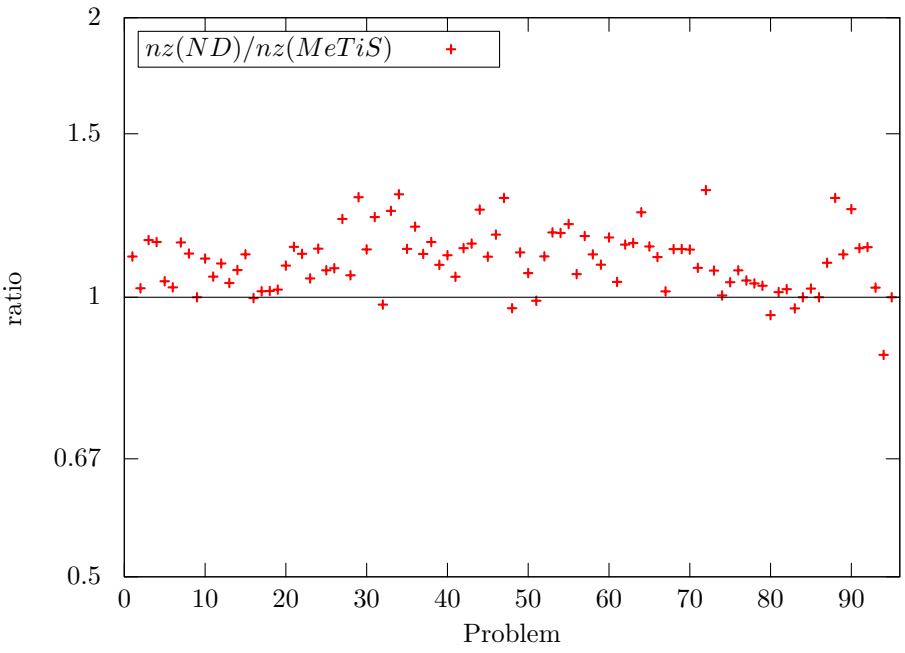

Figure B.14: Comparison of nz for `SPRAL_ND` (denoted by ND) using the multilevel approach with `MeTiS`. The problems are in the order given in Table A.1.

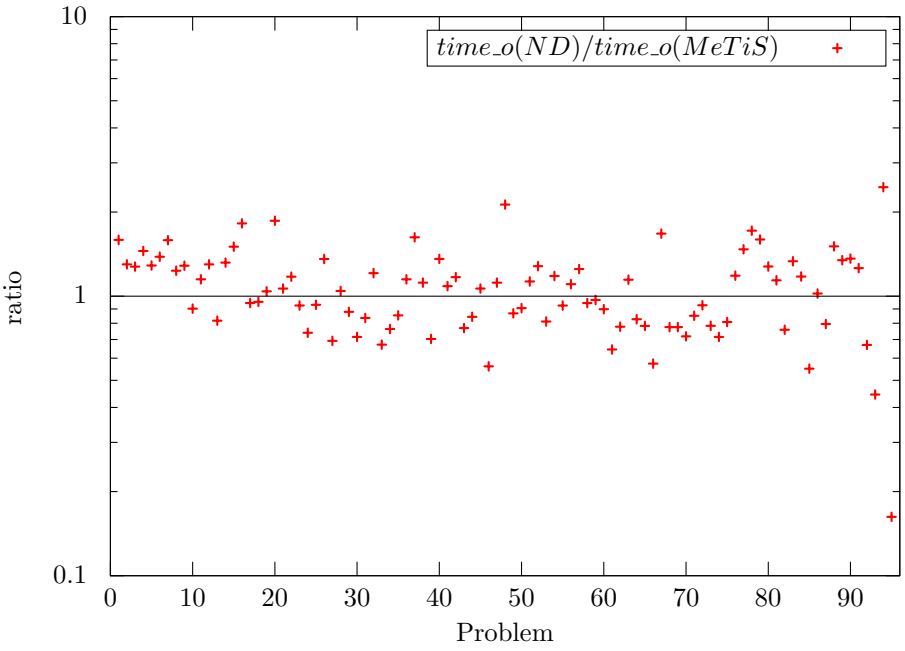

Figure B.15: Comparison of *time_o* for `SPRAL_ND` (denoted by ND) using the multilevel approach with `MeTiS`. The problems are in the order given in Table A.1.

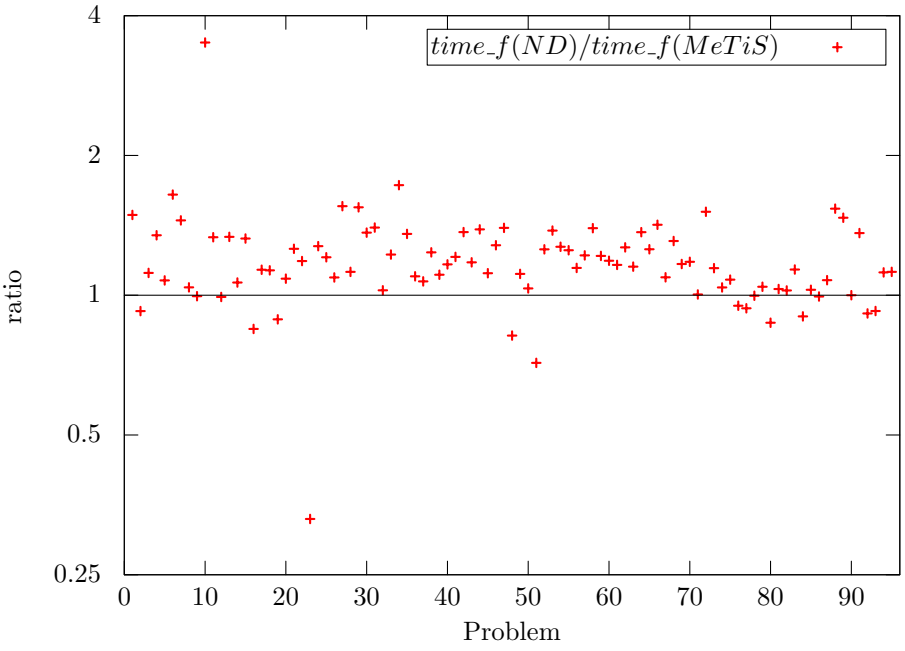

Figure B.16: Comparison of *time_f* for `SPRAL_ND` (denoted by ND) using the multilevel approach with `MeTiS`. The problems are in the order given in Table A.1.

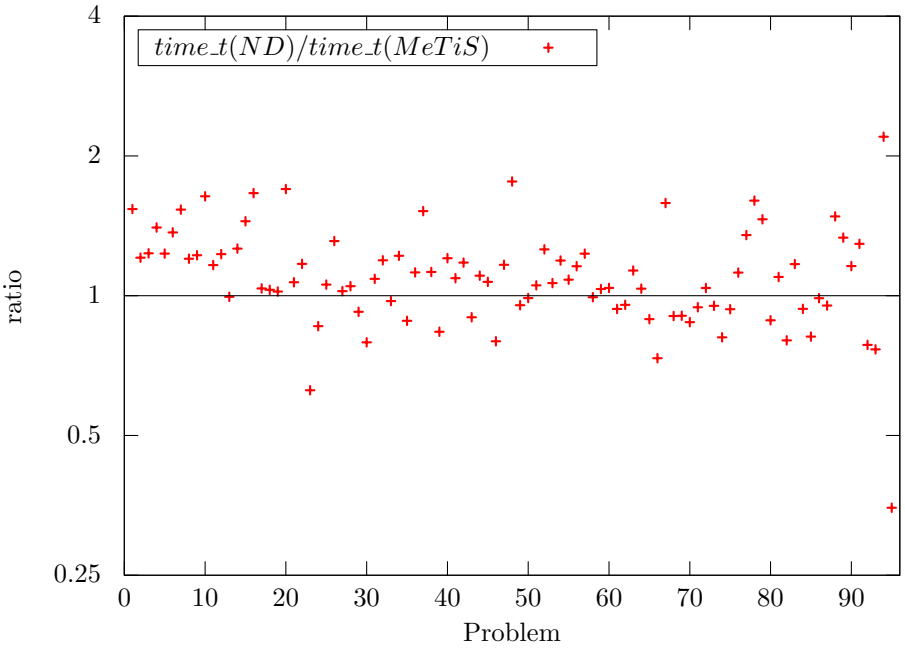

Figure B.17: Comparison of *time_t* for `SPRAL_ND` (denoted by ND) using the multilevel approach with `MeTiS`. The problems are in the order given in Table A.1.