

On computing inverse entries of a sparse matrix in an out-of-core environment¹

Patrick R. Amestoy², Iain S. Duff^{3,4}, Yves Robert⁵, François-Henry Rouet², and Bora Uçar^{5,6}

Technical Report TR/PA/10/59

August 23, 2010

CERFACS
42 Ave G. Coriolis
31057 Toulouse Cedex
France

ABSTRACT

The inverse of an irreducible sparse matrix is structurally full, so that it is impractical to think of computing or storing it. However, there are several applications where a subset of the entries of the inverse is required. Given a factorization of the sparse matrix held in out-of-core storage, we show how to compute such a subset efficiently, by accessing only parts of the factors. When there are many inverse entries to compute, we need to guarantee that the overall computation scheme has reasonable memory requirements, while minimizing the cost of loading the factors. This leads to a partitioning problem that we prove is NP-complete. We also show that we cannot get a close approximation to the optimal solution in polynomial time. We thus need to develop heuristic algorithms, and we propose: (i) a lower bound on the cost of an optimum solution; (ii) an exact algorithm for a particular case; (iii) two other heuristics for a more general case; and (iv) hypergraph partitioning models for the most general setting. We illustrate the performance of our algorithms in practice using the MUMPS software package on a set of real-life problems as well as some standard test matrices. We show that our techniques can improve the execution time by a factor of 50.

Keywords: Sparse matrices, direct methods for linear systems and matrix inversion, multifrontal method, graphs and hypergraphs.

AMS(MOS) subject classifications: 05C50, 05C65, 65F05, 65F50

¹Current reports available at <http://www.cerfacs.fr/algor/reports/index.html>. Also appeared as Technical Report RAL-TR-2010-027 from Rutherford Appleton Laboratory, Oxfordshire. This work was supported by the EPSRC Grant EP/E053351/1 and by “Agence Nationale de la Recherche”, through SOLSTICE project ANR-06-CIS6-010.

²Université de Toulouse, INPT(ENSEEIH)-IRIT, France (`{amestoy,frouet}@enseeiht.fr`).

³Atlas Centre, STFC Rutherford Appleton Laboratory, Oxon OX11 0QX, UK (`i.s.duff@rl.ac.uk`).

⁴CERFACS, 42 Av. G. Coriolis, 31057, Toulouse, France (`duff@cerfacs.fr`).

⁵Laboratoire de l’Informatique du Parallélisme, (UMR CNRS -ENS Lyon-INRIA-UCBL), Université de Lyon, 46, allée d’Italie, ENS Lyon, F-69364, Lyon Cedex 7, France (`yves.robert@ens-lyon.fr`, `bora.ucar@ens-lyon.fr`)

⁶Centre National de la Recherche Scientifique.

Contents

1	Introduction	1
2	Background and problem definition	3
2.1	Elimination tree and sparse triangular solves	3
2.2	Problem definition	7
3	Partitioning methods and models	9
3.1	A partitioning based on post-order	12
3.2	A special case: Two items per part	13
3.3	A heuristic for a more general case	14
3.4	Models based on hypergraph partitioning	14
3.4.1	Hypergraphs and the hypergraph partitioning problem	15
3.4.2	The model	15
4	The off-diagonal case	17
5	Experiments	19
5.1	Assessing the heuristics	19
5.2	Practical tests with a direct solver	20
5.3	Hypergraph model	22
6	Similar problems and related work	25
6.1	In-core case	25
6.2	Parallelism	25
6.3	Related work	26
7	Conclusion	27

1 Introduction

We are interested in efficiently computing entries of the inverse of a large sparse nonsingular matrix. It was proved (Duff, Erisman, Gear and Reid 1988, Gilbert and Liu 1993) that the inverse of an irreducible sparse matrix is full in a structural sense and hence it is impractical to compute all the entries of the inverse. However, there are applications where only a set of entries of the inverse is required. For example, after solving a linear least-squares problem, the variance of each component provides a measure of the quality of the fit, and is given by the diagonal entries of the inverse of a large symmetric positive semi-definite matrix (Björck 1996); the off-diagonal entries of the inverse of the same matrix give the covariance of the components. Other applications arise in quantum-scale device simulation, such as the atomistic level simulation of nanowires (Cauley, Jain, Koh and Balakrishnan 2007, Luisier, Schenk, Fichtner and Klimeck 2006), and in electronic structure calculations (Lin, Lu, Ying, Car and E 2009). In these applications, the diagonal entries of the inverse of a large sparse matrix need to be computed. Some other classical applications which need the entries of the inverse include the computation of short-circuit currents (Takahashi, Fagan and Chin 1973) and approximations of condition numbers (Björck 1996). In all these computational applications, the aim is to compute a large set of entries (often diagonal entries) of the inverse of a large sparse matrix.

We have been particularly motivated by an application in astrophysics which is used by the CESR (Centre for the Study of Radiation in Space, Toulouse). In the context of the INTEGRAL (INTErnational Gamma-Ray Astrophysics Laboratory) mission of ESA (European Space Agency), a spatial observatory with high resolution was launched. SPI (SPectrometer on INTEGRAL), a spectrometer with high energy resolution, is one of the main instruments on board this satellite. To obtain a complete sky survey, a very large amount of data acquired by the SPI must be processed. For example, to estimate the total point-source emission contributions (that is the contribution of a set of sources to the observed field), a linear least-squares problem of about a million equations and a hundred thousand unknowns has to be solved (Bouchet, Roques, Mandrou, Strong, Diehl, Lebrun and Terrier 2005). Once the least-square problem is solved, the variance of the components of the solution are computed to get access to its standard deviation. The variances of the components of the solution are given by the diagonal elements of the inverse of the variance-covariance matrix $A = B^T B$, where B is the matrix associated with the least-squares problem.

The approach we use to compute the entries of the inverse relies on a traditional solution method and makes use of the equation $AA^{-1} = I$. More specifically, we compute a particular entry a_{ij}^{-1} using $(A^{-1}e_j)_i$; here e_j is the j th column of the identity matrix, and $(v)_i$ denotes the i th component of the vector v (we use v_i to refer to the i th component of a vector when v is not defined by an operation). Using an LU factorization of A , a_{ij}^{-1} is obtained by solving successively two triangular systems:

$$\begin{cases} y = L^{-1}e_j \\ a_{ij}^{-1} = (U^{-1}y)_i \end{cases} . \quad (1.1)$$

The computational framework in which we consider this problem assumes that the matrix A , whose inverse entries will be computed, has been factorized using a multifrontal or supernodal approach, and that the factors have been stored on disks (*out-of-core* setting). While solving (1.1),

Table 1.1: All diagonal entries of the inverse are computed in blocks of size 16. The columns “No ES” correspond to solving the linear systems as if the right-hand side vector were dense. The columns “ES” correspond to solving the linear system while exploiting the sparsity of the right-hand side vectors. The computations were performed on the Intel system defined in Section 5.2.

Matrix name	Factors loaded (in MB.)		Time (in secs.)			
			In-core		Out-of-core	
	No ES	ES	No ES	ES	No ES	ES
CESR46799	114051	3158	380	21	3962	77
CESR72358	375737	6056	1090	48	10801	264
CESR148286	1595645	16595	4708	188	43397	721

the computational time is therefore dominated by the time required to load the factors from the disk. We see from the above equations that, in the forward substitution phase, the right-hand side (e_j) contains only one nonzero entry and that, in the backward step, only one entry of the solution vector is required. For efficient computation, we have to take advantage of both these observations along with the sparsity of A . We note that even though the vector y will normally be sparse, it will conventionally be stored as a full dense vector. Therefore, when the number of requested entries is high, one cannot hold all the solution vectors in the memory at the same time. In this case, the computations are carried out in epochs where, at each epoch, a predefined number of requested entries are computed. Table 1.1 contains results for three medium-sized matrices from our motivating application and summarizes what can be gained using our approach (the details of the experimental setting are described later in Section 5.2). In this table, we show the factors loaded and the execution time of the MUMPS (MUltifrontal Massively Parallel Solver) software package (Amestoy, Duff, Koster and L’Excellent 2001, Amestoy, Guermouche, L’Excellent and Pralet 2006): we compare the memory requirement and solution time when sparsity in both the right-hand side and the solution is not exploited (in column No ES), and when it is exploited (in column ES). The solution time is shown for both the out-of-core and the in-core case, in order to put these execution times in perspective. All diagonal entries of the inverse are computed, where 16 entries are computed at each epoch. Both solution schemes use the natural partition that orders the requested entries according to their indices, and puts them in blocks of size 16 in a natural order, that is the first 16 in the first block, the second 16 in the second block, and so on. We see that, on this class of problems, the number of factors to be loaded is reduced by a factor of between 36 and 96, with an impact of the same order on the computation time. Our aim in this paper is to further reduce the factors loaded and the execution time by carefully partitioning the requested entries.

In Section 2.1, we formulate which parts of the factors need to be loaded for a requested entry. At this stage, our contribution is to summarize what exists in the literature although we believe that our observations on the solution of $(Ux = y)_i$ using the tree are original. Later on, we investigate the case where a large number of entries of the inverse have been requested. This would be the typical case in applications when we have to proceed in epochs. This entails accessing (loading) some parts of L and U many times in different epochs, according to the entries computed in the corresponding epochs. The main combinatorial problem then becomes that of

partitioning the requested entries into blocks in such a way that the overall cost of loading the factors summed over the epochs is minimized. As we shall see in Section 2.2, the problem can be cast as a partitioning problem on trees. In Section 3, we prove that the partitioning problem is NP-complete, and that a simple post-order based heuristic gives a solution whose cost is at most twice the cost of an optimal solution. We propose an exact algorithm for a particular case of the partitioning problem, and use this algorithm to design a heuristic for a more general case. We also propose a hypergraph-partitioning formulation which provides a solution to the most general problem of computing off-diagonal entries (see Section 4). We illustrate the performance of our techniques by implementing them within MUMPS, both on standard test matrix problems, and on specific examples coming from data fitting problems in astrophysics. We present our results in Section 5. Some problems that are related to those covered in this paper are discussed in Section 6, along with directions for further research. We provide some concluding remarks in Section 7.

2 Background and problem definition

Throughout this study, we use a direct factorization method for obtaining the inverse entries. We assume that the factorization has taken place, and that the factors are stored on the disks. The factorization code that we use is the general-purpose linear solver MUMPS that computes the factorization $A = LU$ if the matrix is unsymmetric, or the factorization $A = LDL^T$ if the matrix is symmetric. MUMPS provides a large range of functionalities, including the ability to run *out-of-core*, that is it can store the factors on hard disk when the main memory is not large enough (Agullo 2008, Slavova 2009). We consider an out-of-core environment and the associated metrics. We assume that the factorization has taken place using the structure of $A + A^T$ for a pattern unsymmetric A , where the summation is structural. In this case, the structures of the factors L and U are the transpose of each other. All computation schemes, algorithms, and formulations in this paper assume this last property to hold.

We need the following standard definitions for later. A *topological ordering* of a tree is an ordering of its nodes such that any node is numbered before its parent. A *post-order* of a tree is a topological ordering where the nodes in each subtree are numbered consecutively. The least common ancestor of two nodes i and j in a rooted tree, $lca(i, j)$, is the lowest numbered node that lies at the intersection of the unique paths from node i and node j to the root. The ceiling function $\lceil x \rceil$ gives the smallest integer greater than or equal to x . For a set S , $|S|$ denotes the cardinality of S .

2.1 Elimination tree and sparse triangular solves

When the matrix A is structurally symmetric with a zero-free diagonal, the *elimination tree* represents the storage and computational requirements of its sparse factorization. There are a few equivalent definitions of the elimination tree (Liu 1990). We prefer the following one for the purposes of this paper:

Definition 2.1 *Assume $A = LU$ where A is a sparse, structurally symmetric, $N \times N$ matrix. Then, the elimination tree $T(A)$ of A is a tree of N nodes, with the i th node corresponding to the*

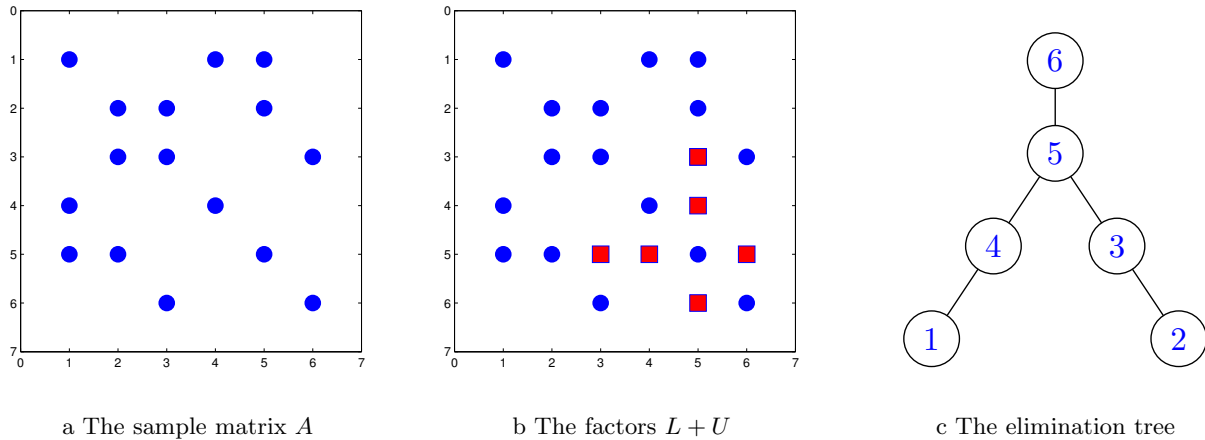


Figure 2.1: A pattern symmetric matrix A its factors and the associated elimination tree. (a) The matrix A whose nonzeros are shown with blue circles, (b) The pattern of $L + U$ where the filled-in entries are shown with red squares, (c) The corresponding elimination tree where the children of a node are drawn below the node itself.

i th column of L , and where the parent relations are defined as follows:

$$\text{parent}(j) = \min\{i : i > j \text{ and } \ell_{ij} \neq 0\} \text{ for } j = 1, \dots, N - 1.$$

For the sake of completeness we note that if A is reducible, this structure is a forest with one tree for each irreducible block; otherwise it is a tree. We assume without loss of generality that the matrix is irreducible. As an example consider the pattern symmetric matrix A shown in Fig. 1a. The factors and the corresponding elimination tree are shown in Figs. 1b and 1c.

Our algorithms for efficiently computing a given set of entries in A^{-1} rely on the elimination tree structure. We take advantage of the following result, which is rewritten from Gilbert and Liu (1993, Theorem 2.1).

Corollary 2.1 *Assume b is a sparse vector and L is a lower triangular matrix. Then the indices of the nonzero elements of the solution vector x of $Lx = b$ are equal to the indices of the nodes of the elimination tree that are in the paths from the nodes corresponding to nonzero entries of b to the root.*

We will use the corollary in the following way: when b is sparse, we need to solve the equations corresponding to the predicted nonzero entries, while setting the other entries of the solution vector x to zero. We note that when b contains a single nonzero entry, say in its i th component, the equations to be solved are those that correspond to the nodes in the unique path from node i to the root. In any case, assuming the matrix is irreducible, the last entry x_N , corresponding to the root of the tree, is nonzero. Consider for example $Lx = e_3$ for the lower triangular factor L shown in Fig. 1b. Clearly, $x_1, x_2 = 0$ and $x_3 = 1/\ell_{33}$. These in turn imply that $x_4 = 0$ and $x_5, x_6 \neq 0$. The nonzero entries correspond to the nodes that are in the unique path from node 3 to the root node 6, as seen in Fig. 1c.

With Corollary 2.1, we are halfway through solving (1.1) efficiently for a particular entry of the inverse ; we only solve only the relevant equations involving the L factor (and hence only

access the necessary parts of it). Next, we show which equations involving U need to be solved when only a particular entry of the solution is requested, thereby specifying the whole solution process for an entry of the inverse.

Lemma 2.2 *In order to obtain the i th component of the solution to $Uz = y$, that is $z_i = (U^{-1}y)_i$, one has to solve the equations corresponding to the nodes that are in the unique path from the highest node in $struct(y) \cap ancestors(i)$ to i , where $struct(y)$ denotes the nodes associated with the nonzero entries of y , and $ancestors(i)$ denotes the sets of ancestors of node i in the elimination tree.*

Proof. We first show (1) that the set of components of z involved in the computation of z_i is the set of ancestors of i in the elimination tree. We then (2) reduce this set using the structure of y .

(1) We prove, by top-down induction on the tree, that the only components involved in the computation of any component z_l of z are the ancestors of l in the elimination tree:

- Root node: z_N is computed as $z_N = y_N/u_{NN}$ (thus requires no other component of z) and has no ancestor in the tree.
- For any node l : following a left-looking scheme, z_l is computed as:

$$z_l = \left(y_l - \sum_{k=l+1}^N u_{lk}z_k \right) / u_{ll} = \left(y_l - \sum_{k:u_{lk} \neq 0}^N u_{lk}z_k \right) / u_{ll} \quad (2.1)$$

All the nodes in the set $\mathcal{K}_l = \{k : u_{lk} \neq 0\}$ are ancestors of l by definition of the elimination tree (since $struct(U^T) = struct(L)$). Thus, by applying the induction hypothesis to all the nodes in \mathcal{K}_l , all the required nodes are ancestors of l .

(2) The pattern of y can be exploited to show that some components of z which are involved in the computation of z_i are zero. Noting k_i as the highest node in $struct(y) \cap ancestors(i)$, that is the highest ancestor of i such that $y_{k_i} \neq 0$, we have:

$$\begin{cases} z_k = 0 & \text{if } k \in ancestors(k_i) \\ z_k \neq 0 & \text{if } k \in ancestors(i) \setminus ancestors(k_i) . \end{cases}$$

Both statements are proved by induction using the same left-looking scheme.

Therefore, the only components of z required lie on the path between i and k_i , the highest node in $struct(y) \cap ancestors(i)$. \square

In particular, when y_N is nonzero, as would be the case when y is the vector obtained after forward elimination for the factorization of an irreducible matrix, Lemma 2.2 states that we need to solve the equations that correspond to nodes that lie in the unique path from the root node to node i . Consider the U given in Fig. 1b, and suppose we want to compute $(Uz = e_6)_2$. As we are interested in z_2 , we have to compute z_3 and z_5 to be able to solve the second equation; in order to compute z_3 we have to compute z_6 as well. Therefore, we have to solve equation 2.1 for nodes 2, 3, 5, and 6. As seen in Fig. 1c, these correspond to the nodes that are in the unique path

from node 6 (the highest ancestor) to node 2. We also note that variable z_4 would be nonzero, however we will not compute it, because it does not play a role in determining z_2 .

One can formulate a_{ij}^{-1} in three different ways, each involving linear system solutions with L and U . Consider the general formula:

$$\begin{aligned} a_{ij}^{-1} &= e_i^T A^{-1} e_j \\ &= e_i^T U^{-1} L^{-1} e_j, \end{aligned} \tag{2.2}$$

and the three possible ways of parenthesizing these equations. Our method as shown in (1.1) corresponds to the parenthesization

$$(U^{-1} (L^{-1} e_j))_i. \tag{2.3}$$

The other two parenthesizations are:

$$a_{ij}^{-1} = ((e_i^T U^{-1}) L^{-1})_j \tag{2.4}$$

$$= (e_i^T U^{-1}) (L^{-1} e_j). \tag{2.5}$$

We have the following theorem regarding the equivalence of these three parenthesizations, when L and U are computed and stored in such way that their sparsity patterns are the transposes of each other. In a more general case, the three parenthesizing schemes may differ when L and U^T have different patterns.

Theorem 2.3 *The three parenthesizations for computing a_{ij}^{-1} given in equations (2.3)-to-(2.5) access the same parts of the factor L and the same parts of the factor U .*

Proof. Consider the following four computations on which the different parenthesizations are based:

$$v^T = e_i^T U^{-1} \tag{2.6}$$

$$w_i = (U^{-1} y)_i \tag{2.7}$$

$$y = L^{-1} e_j \tag{2.8}$$

$$z_j = (v L^{-1})_j. \tag{2.9}$$

As $v = U^{-T} e_i$ and the pattern of U^T is equal to the pattern of L , by Corollary 2.1, computing v requires accessing the U factors associated with nodes in the unique path between node i and the root node. Consider $z^T = L^{-T} v^T$. As the pattern of L^T is equal to the pattern of U , by Lemma 2.2, z_j requires accessing the L factors associated with the nodes in the unique path from node j to the highest node in $struct(v) \cap ancestors(j)$. As $v_N = (U^{-T} e_i)_N \neq 0$ (since A is irreducible) this requires accessing the L factors from node j to the root (the N th node). With similar arguments, $y_N \neq 0$, and computing $w_i = (U^{-1} y)_i$ requires accessing the U factors associated with the nodes in the unique path between node i and the root. These observations combined with another application of Corollary 2.1, this time for $L^{-1} e_j$, complete the proof. \square

We now combine Corollary 2.1 and Lemma 2.2 to obtain the theorem that is Property 8.9 in Slavova (2009):

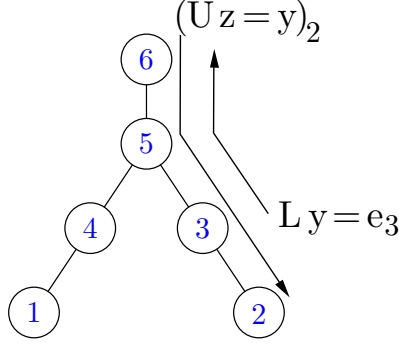


Figure 2.2: Traversal of the elimination tree for the computation of a_{23}^{-1} . In the first step $Ly = e_3$ is solved yielding y_3, y_5 , and $y_6 \neq 0$. Then $(Uz = y)_2$ is found by computing z_6, z_5, z_3 , and finally z_2 .

Theorem 2.4 (Factors to load for computing a particular entry of the inverse) *To compute a particular entry a_{ij}^{-1} in A^{-1} , the only factors which have to be loaded are the L factors on the path from node j up to the root node, and the U factors on the path going back from the root to node i .*

This theorem establishes the efficiency of the proposed computation scheme: we solve only the equations required for a requested entry of the inverse, both in the forward and backward solve phases. We illustrate the above theorem on the previous example in Fig. 2.2. As discussed before, $Ly = e_3$ yields nonzero vector entries y_3, y_5 , and y_6 , and then $z_2 = (Uz = y)_2$ is found after computing z_6, z_5, z_3 .

We note that the third parenthesization (2.5) can be advantageous while computing only the diagonal entries with the factorizations LL^T or LDL^T (with a diagonal D matrix), because, in these cases, we need to compute a single vector and compute the square of its norm. This formulation can also be useful in a parallel setting where the solves with L and U can be computed in parallel, whereas in the other two formulations, the solves with one of the factors has to wait for the solves with the other one to be completed.

We also note that if the triangular solution procedures for $e_i^T U^{-1}$ and $L^{-1} e_j$ are available, then one can benefit from the third parenthesization in certain cases. If the number of row and column indices concerned by the requested entries is smaller than the number of these requested entries, many of the calculations can be reused if one computes a set of vectors of the form $e_i^T U^{-1}$ and $L^{-1} e_j$ for different i and j and obtains the requested entries of the inverse by computing the inner products of these vectors. We do not consider this computational scheme in this paper because such separate solves with L and U are not available within MUMPS.

2.2 Problem definition

We now address the computation of multiple entries, and introduce the partitioning problem. We first discuss the diagonal case (that is computing a set of diagonal entries), and comment on the general case later in Section 4.

As seen in the previous section, in order to compute a_{ii}^{-1} using the formulation

$$\begin{cases} y = L^{-1}e_i \\ a_{ii}^{-1} = (U^{-1}y)_i \end{cases}$$

we have to access the parts of L that correspond to the nodes in the unique path from node i to the root, and then access the parts of U that correspond to the nodes in the same path. As discussed above, these are the necessary and sufficient parts of L and U that are needed. In other words, we know how to solve efficiently for a single requested diagonal entry of the inverse. Now suppose that we are to compute a set R of diagonal entries of the inverse. As said in Section 1, using the equations above entails storing a dense vector for each requested entry. If $|R|$ is small, then we could again identify all the parts of L and U that are needed to be loaded at least for one requested entry in R and then solve for all R at once, accessing the necessary and sufficient parts of L and U only once. However, $|R|$ is usually large: in the application areas mentioned in Section 1, one often wants to compute a large set of entries, such as the whole diagonal of the inverse (in that case, $|R| = N$). Storing that many dense vectors is not feasible; therefore, the computations proceed in epochs, where at each epoch a limited number of diagonal entries are computed. This entails accessing (loading) some parts of L and U multiple times in different epochs according to the entries computed in the corresponding epochs. The main combinatorial problem then becomes that of partitioning the requested entries into blocks in such a way that the overall cost of loading the factors is minimized.

We now formally introduce the problem. Let T be the elimination tree on N nodes, where the factors associated with each node are stored on disks (*out-of-core*). Let $P(i)$ be the set of nodes in the unique path from node i to the root r , including both nodes i and r . Let $w(i)$ denote the cost of loading the parts of the factors, L or U , associated with node i of the elimination tree. Similarly let $w(i, j)$ denote the sum of the costs of the nodes in the path from node i to node j . The cost of solving a_{ii}^{-1} is therefore

$$cost(i) = \sum_{k \in P(i)} 2 \times w(k) = 2 \times w(i, r). \quad (2.10)$$

If we solve for a set R of diagonal entries at once, then the overall cost is therefore

$$cost(R) = \sum_{i \in P(R)} 2 \times w(i) \text{ where } P(R) = \bigcup_{i \in R} P(i).$$

We use B to denote the maximum number of diagonal entries that can be computed at an epoch. This is the number of dense vectors that we must hold and so is limited by the available storage.

The TREEPARTITIONING problem is formally defined as follows: given a tree T with N nodes, a set $R = \{i_1, \dots, i_m\}$ of nodes in the tree, and an integer $B \leq m$, partition R into a number of subsets R_1, R_2, \dots, R_K so that $|R_k| \leq B$ for all k , and the total cost

$$cost(R) = \sum_{k=1}^K cost(R_k) \quad (2.11)$$

is minimum.

The number of subsets K is not specified, but obviously $K \geq \lceil \frac{m}{B} \rceil$. Without loss of generality we can assume that there is a one-to-one correspondence between R and leaf nodes in T . Indeed,

if there is a leaf node i where $i \notin R$, then we can delete node i from T . Similarly, if there is an internal node i where $i \in R$, then we create a leaf node i' of zero weight ($w_{i'} = 0$) and make it an additional child of i . For ease of discussion and formulation, for each requested node (leaf or not) of the elimination tree we add a leaf node with zero weight. To clarify the execution scheme, we now specify the algorithm that computes the diagonal entries of the inverse specified by a given R_k . We first find $P(R_k)$; we then post-order the nodes in $P(R_k)$, and start loading the associated L factors from the disk and perform the forward solves with L . When we reach the root node, we have $|R_k|$ dense vectors and we start loading the associated U factors from the disk and perform backward substitutions along the paths that we traversed (in reverse order) during the forward substitutions.

3 Partitioning methods and models

As discussed above, partitioning the requested entries into blocks to minimize the cost of loading factors corresponds to the TREEPARTITIONING problem. In this section, we will focus on the case where all of the requested entries are on the diagonal of the inverse. As noted before, in this case, the partial forward and backward solves correspond to visiting the same path in the elimination tree. We analyse the TREEPARTITIONING problem in detail for this case and show that it is NP-complete; we also show that the case where the block size $B = 2$ is polynomial time solvable. We provide two heuristics, one with an approximation guarantee (in the sense that we can prove that it is at worst twice as bad as optimal), and the other being somewhat better in practice; we also introduce a hypergraph partitioning-based formulation which is more general than the other heuristics.

Before introducing the algorithms and models, we present a lower bound for the cost of an optimal partition. Let $nl(i)$ denote the number of leaves of the subtree rooted at node i which can be computed as follows:

$$nl(i) = \begin{cases} 1 & i \text{ is a leaf node,} \\ \sum_{j \in \text{children}(i)} nl(j) & \text{otherwise.} \end{cases} \quad (3.1)$$

We note that as all the leaf nodes correspond to the requested diagonal entries of the inverse, $nl(i)$ corresponds to the number of forward and backward solves that have to be performed at node i .

Given the number of forward and backward solves that pass through a node i , it is easy to define the following lower bound on the amount of the factors loaded.

Theorem 3.1 (Lower bound of the amount of factors to load) *Let T be a node weighted tree, $w(i)$ be the weight of node i , B be the maximum allowed size of a partition, and $nl(i)$ be the number of leaf nodes in the subtree rooted at i . Then we have the following lower bound, denoted by η , on the optimal solution c^* of the TREEPARTITIONING problem:*

$$\eta = 2 \times \sum_{i \in T} w(i) \times \left\lceil \frac{nl(i)}{B} \right\rceil \leq c^* .$$

Proof. Follows easily by noting that each node i has to be loaded at least $\left\lceil \frac{nl(i)}{B} \right\rceil$ times. \square

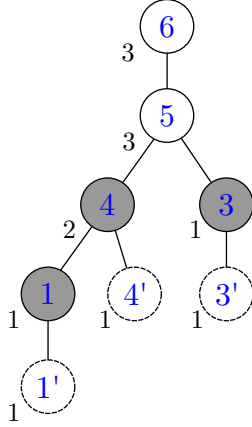


Figure 3.1: Number of leaves of the subtrees rooted at each node of a transformed elimination tree. The nodes corresponding to the requested diagonal entries of the inverse are shaded, and a leaf node is added for each such entry. Each node is annotated with the number of leaves in the corresponding subtree; resulting in a lower bound of $\eta = 14$ with $B = 2$.

As the formula includes $nl(\cdot)$, the lower bounds for wide and shallow trees will usually be smaller than the lower bounds for tall and skinny trees. Each internal node is on a path from (at least) one leaf node, therefore $\lceil nl(i)/B \rceil$ is at least 1 and $2 \times \sum_i w(i) \leq c^*$.

Figure 3.1 illustrates the notion of the number of leaves of a subtree and the computation of the lower bound. Entries a_{11}^{-1} , a_{33}^{-1} , and a_{44}^{-1} are requested, and the elimination tree of Figure 1c is modified accordingly to have leaves corresponding to these entries. The numbers $nl(i)$ are shown next to the nodes. Suppose that each node has unit weight and that the block size is 2. Then, the lower bound is:

$$\begin{aligned} \eta &= 2 \times \left(\left\lceil \frac{1}{2} \right\rceil + \left\lceil \frac{1}{2} \right\rceil + \left\lceil \frac{2}{2} \right\rceil + \left\lceil \frac{3}{2} \right\rceil + \left\lceil \frac{3}{2} \right\rceil \right) \\ &= 14. \end{aligned}$$

Recall that we have transformed the elimination tree in such a way that the requested entries now correspond to the leaves and each leaf corresponds to a requested entry. We have the following computational complexity result.

Theorem 3.2 *The TREEPARTITIONING problem is NP-complete.*

Proof. We consider the associated decision problem: given a tree T with m leaves, a value of B , and a cost bound c , does there exist a partitioning S of the m leaves into subsets whose size does not exceed B , and such that $cost(S) \leq c$? It is clear that this problem belongs to NP since if we are given the partition S , it is easy to check in polynomial time that it is valid and that its cost meets the bound c . We now have to prove that the problem is in the NP-complete subset.

To establish the completeness, we use a reduction from 3-PARTITION (Garey and Johnson 1979), which is NP-complete in the strong sense. Consider an instance \mathcal{I}_1 of 3-PARTITION: given a set $\{a_1, \dots, a_{3p}\}$ of $3p$ integers, and an integer Z such that $\sum_{1 \leq j \leq 3p} a_j = pZ$, does there exist a partition of $\{1, \dots, 3p\}$ into p disjoint subsets K_1, \dots, K_p , each with three elements, such that for all $1 \leq i \leq p$, $\sum_{j \in K_i} a_j = Z$?

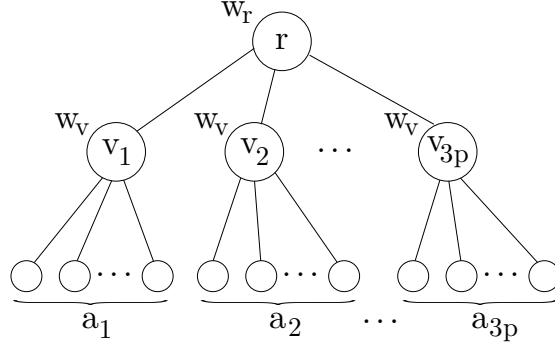


Figure 3.2: The instance of the TREEPARTITIONING problem corresponding to a given 3-PARTITION PROBLEM. The weight of each node is shown next to the node. The minimum cost of a solution for $B = Z$ to the TREEPARTITIONING problem is $p \times w_r + 3p \times w_v$ which is only possible when the children of each v_i are all in the same part, and when the children of three different internal nodes, say v_i, v_j, v_k , are put in the same part. This corresponds to putting the numbers a_i, a_j, a_k into a set for the 3-PARTITION problem which sums up to Z .

We build the following instance \mathcal{I}_2 of our problem: the tree is a three-level tree composed of $N = 1 + 3p + pZ$ nodes: the root v_r , of cost w_r , has $3p$ children v_i , of same cost w_v , for $1 \leq i \leq 3p$. In turn, each v_i has a_i children, each being a leaf node of zero cost. This instance \mathcal{I}_2 of the TREEPARTITIONING problem is shown in Fig. 3.2. We let $B = Z$ and ask whether there exists a partition of leaf nodes of cost $c = pw_r + 3pw_v$. Here w_r and w_v are arbitrary values (we can take $w_r = w_v = 1$). We note that the cost c corresponds to the lower bound shown in Theorem 3.1; in this lower bound, each internal node v_i is loaded only once, and the root is loaded p times, since it has $pZ = pB$ leaves below it. Note that the size of \mathcal{I}_2 is polynomial in the size of \mathcal{I}_1 . Indeed, because 3-PARTITION is NP-complete in the strong sense, we can encode \mathcal{I}_1 in unary, and the size of the instance is $O(pZ)$.

Now we show that \mathcal{I}_1 has a solution if and only if \mathcal{I}_2 has a solution. Suppose first that \mathcal{I}_1 has a solution K_1, \dots, K_p . The partition of leaf nodes corresponds exactly to the subsets K_i : we build p subsets S_i whose leaves are the children of vertices v_j with $j \in K_i$. Suppose now that \mathcal{I}_2 has a solution. To meet the cost bound, each internal node has to be loaded only once, and the root at most p times. This means that the partition involves at most p subsets to cover all leaves. Because there are pZ leaves, each subset is of size exactly Z . Because each internal node is loaded only once, all its leaves belong to the same subset. Altogether, we have found a solution to \mathcal{I}_1 , which concludes the proof. \square

We can further show that we cannot get a close approximation to the optimal solution in polynomial time.

Theorem 3.3 *Unless $P=NP$, there is no $1 + o(\frac{1}{N})$ polynomial approximation for trees with N nodes in the TREEPARTITIONING problem.*

Proof. Assume that there exists a polynomial $1 + \frac{\varepsilon(N)}{N}$ approximation algorithm for trees with N nodes, where $\lim_{N \rightarrow \infty} \varepsilon(N) = 0$. Let $\varepsilon(N) < 1$ for $N \geq N_0$. Consider an arbitrary instance \mathcal{I}_0 of 3-PARTITION with a set $\{a_1, \dots, a_{3p}\}$ of $3p$ integers, and an integer Z such that $\sum_{1 \leq j \leq 3p} a_j = pZ$. Without loss of generality, assume that $a_i \geq 2$ for all i (hence $Z \geq 6$). We

ask if we can partition the $3p$ integers of \mathcal{I}_0 into p triples of the same sum Z . Now we build an instance \mathcal{I}_1 of 3-PARTITION by adding X times the integer $Z - 2$ and $2X$ times the integer 1 to \mathcal{I}_0 , where $X = \max\left(\left\lceil \frac{N_0 - 1}{Z + 3} \right\rceil - p, 1\right)$. Hence \mathcal{I}_1 has $3p + 3X$ integers and we ask whether these can be partitioned into $p + X$ triples of the same sum Z . Clearly, \mathcal{I}_0 has a solution if and only if \mathcal{I}_1 does (the integer $Z - 2$ can only be in a set with two 1s).

We build an instance \mathcal{I}_2 of TREEPARTITIONING from \mathcal{I}_1 exactly as we did in the proof of Theorem 3.2, with $w_r = w_v = 1$, and $B = Z$. The only difference is that the value p in the proof has been replaced by $p + X$ here, therefore the three-level tree now has $N = 1 + 3(p + X) + (p + X)Z$ nodes. Note that X has been chosen so that $N \geq N_0$. Just as in the proof of Theorem 3.2, \mathcal{I}_1 has a solution if and only if the optimal cost for the tree is $c^* = 4(p + X)$, and otherwise the optimal cost is at least $4(p + X) + 1$.

If \mathcal{I}_1 has a solution, and because $N \geq N_0$, the approximation algorithm will return a cost at most

$$\left(1 + \frac{\varepsilon(N)}{N}\right) c^* \leq \left(1 + \frac{1}{N}\right) 4(p + X) = 4(p + X) + \frac{4(p + X)}{N}$$

But $\frac{4(p+X)}{N} = \frac{4(N-1)}{(Z+3)N} \leq \frac{4}{9} < 1$, so that the approximation algorithm can be used to determine whether \mathcal{I}_1 , and hence \mathcal{I}_0 , has a solution. This is a contradiction unless $P=NP$. \square

3.1 A partitioning based on post-order

Consider again the case where some entries in the diagonal of the inverse are requested. As said before, the problem of minimizing the size of the factors to be loaded corresponds to the TREEPARTITIONING problem. Consider the heuristic POPART shown in Algorithm 1 for this problem.

Algorithm 1 POPART: A post-order based partitioning

Input $T = (V, E, r)$ with F leaves; each requested entry corresponds to a leaf node with a zero weight.

Input B : the maximum allowable size of a part.

Output $\Pi_{PO} = \{R_1, \dots, R_K\}$ where $K = \lceil F/B \rceil$, a partition on the leaf nodes.

- 1: compute a post-order
 - 2: $\mathcal{L} \leftarrow$ sort the leaf nodes according to their rank in post-order
 - 3: $R_k = \{\mathcal{L}(i) : (k - 1) \times B + 1 \leq i \leq \min\{k \times B, F\}, \text{ for } k = 1, \dots, \lceil F/B \rceil\}$
-

As seen in Algorithm 1, the POPART heuristic first orders the leaf nodes according to their post-order. It then puts the first B leaves in the first part, the next B leaves in the second part, and so on. This simple partitioning approach results in $\lceil F/B \rceil$ parts, for a tree with F leaf nodes, and puts B nodes in each part, except maybe in the last one. We have the following theorem which states that this simple heuristic obtains results that are at most twice the cost of an optimum solution.

Theorem 3.4 *Let Π_{PO} be the partition obtained by the algorithm POPART and c^* be the cost of an optimum solution, then*

$$\text{cost}(\Pi_{PO}) \leq 2 \times c^* .$$

Proof. Consider node i . Because the leaves of the subtree rooted at i are sorted consecutively in \mathcal{L} , the factors of node i will be loaded at most $\left\lceil \frac{nl(i)}{B} \right\rceil + 1$ times. Therefore the overall cost is at most

$$\begin{aligned} \text{cost}(\Pi_{PO}) &\leq 2 \times \sum_i w(i) \times \left(\left\lceil \frac{nl(i)}{B} \right\rceil + 1 \right) \\ &\leq \eta + 2 \times \sum_i w(i) \\ &\leq 2 \times c^* \quad \square \end{aligned}$$

We note that the factor two in the approximation guarantee would be rather loose in practical settings, as $\sum_i w(i)$ would be much smaller than the lower bound η with a practical B and a large number of nodes.

3.2 A special case: Two items per part

In this section, we propose algorithms to solve the partitioning problem exactly when $B = 2$ so that we are able to use a matching to define the epochs. These algorithms will serve as a building block for $B = 2^k$ in the next subsection.

One of the algorithms is based on graph matching as partitioning into blocks of size 2 can be described as a matching. Consider the complete graph $G = (V, V \times V)$ of the leaves of a given tree, and assume that the edge (i, j) represents the decision to put the leaf nodes i and j together in a part. Given this definition of the vertices and edges, we associate the value $m(i, j) = \text{cost}(\{i, j\})$ to the edge (i, j) if $i \neq j$, and $m(i, i) = \sum_{n \in V} w(n)$ (or any sufficiently large number). Then a minimum weighted matching in G defines a partitioning of the vertices in V with the minimum cost (as defined in (2.11)). Although this is a short and immediate formulation, it has a high run time complexity of $O(|V|^{5/2})$ and $O(|V|^2)$ memory requirements. Therefore, we propose yet another exact algorithm for $B = 2$.

The proposed algorithm MATCH proceeds from the parents of the leaf nodes to the root. At each internal node n , those leaf nodes that are in the subtree rooted at n and which are not put in a part yet, are matched two by two (arbitrarily) and each pair is put in a part; if there is an odd number of leaf nodes remaining to be partitioned at node n , one of them (arbitrarily) is passed to $\text{parent}(n)$. Clearly, this algorithm attains the lower bound shown in Theorem 3.1, and hence it finds an optimal partition for $B = 2$. The memory and the run time requirements are $O(|V|)$. We note that two leaf nodes can be matched only at their least common ancestor.

We have slightly modified the basic algorithm and show this modified version in Algorithm 2. The modifications keep the running time and memory complexities the same, and they are realized to enable the use of MATCH as a building block for a more general heuristic. In this algorithm, $\text{parent}(n)$ gives the parent of node n , and $\text{list}(n)$ is an array associated with node n . The sum of the sizes of the $\text{list}(\cdot)$ arrays is $|V|$. The modification is that, when there are an odd number of leaf nodes to partition at node n , the leaf node with the least cumulative weight is passed to the father. The cumulative weight of a leaf node i when MATCH processes node n is defined as $w(i, n) - w(n)$: the sum of the weights of the nodes in the unique path between nodes i and n , including i , but excluding n . This is easy to compute: each time a leaf node is relayed to the father of the current node, the weight of the current node is added to the cumulative weight of

Algorithm 2 MATCH: An exact algorithm for $B = 2$

Input $T = (V, E, r)$ with the root r and F leaves; each requested entry corresponds to a leaf node with a zero weight.

Output $\Pi_2 = \{R_1, \dots, R_K\}$ where $K = \lceil F/2 \rceil$

```

1: for each leaf node  $\ell$  do
2:   add  $\ell$  to  $\text{list}(\text{parent}(\ell))$ ,
3:   compute a postorder
4:  $k \leftarrow 1$ 
5: for each non-leaf node  $n$  in postorder do
6:   if  $n \neq r$  and  $\text{list}(n)$  contains an odd number of vertices then
7:      $\ell \leftarrow$  the node with least weight in  $\text{list}(n)$ 
8:     move  $\ell$  to  $\text{list}(\text{parent}(n))$ , add  $w(n)$  to the weight of  $\ell$ 
       /* relay it to the father */
9:   else if  $n = r$  and  $\text{list}(r)$  contains an odd number of vertices then
10:     $\ell \leftarrow$  a node with the least weight in  $\text{list}(n)$ 
11:    make  $\ell$  a singleton
12:   for  $i = 1$  to  $|\text{list}(n)|$  by 2 do
13:     Put the  $i$ th and  $i + 1$ st vertices in  $\text{list}(n)$  into  $R_k$ , increment  $k$ 
       /* Match the  $i$ th and  $i + 1$ st elements in the list */

```

the relayed leaf node (as shown in line 8). By doing this, the leaf nodes which traverse longer paths before being partitioned are chosen before those with smaller weights.

3.3 A heuristic for a more general case

We propose a heuristic algorithm when $B = 2^k$ for some k : the BISEMATCH algorithm is shown in Algorithm 3. It is based on a bisection approach. At each bisection, a matching among the leaf nodes is found by a call to MATCH. Then, one of the leaf nodes of each pair is removed from the tree; the remaining one becomes a representative of the two and is called a principal node. Since the remaining node at each bisection step is a representative of the two representative nodes of the previous bisection step, after $\log B = k$ steps, BISEMATCH obtains nodes that represent at most $B - 1$ other nodes. At the end, the nodes represented by a principal node are included in the same part as the principal node.

As seen in the algorithm, at each stage a matching among the remaining leaves is found by using the MATCH algorithm. When leaf nodes i and j are matched at their least common ancestor $\text{lca}(i, j)$, if $w(i, \text{lca}(i, j)) \geq w(j, \text{lca}(i, j))$, then we designate i to be the representative of the two by adding (i, j) to M , otherwise we designate j to be the representative by adding (j, i) to M . With this choice, the MATCH algorithm is guided to make decisions at deeper parts of the tree. The running time of BISEMATCH is $O(|V| \log B)$, with an $O(|V|)$ memory requirement.

3.4 Models based on hypergraph partitioning

We show how the problem of finding an optimal partition of the requested entries can be transformed into a hypergraph partitioning problem. Our aim is to develop a general model that can address both the diagonal and the off-diagonal cases. Here, we give the model again for

Algorithm 3 BISEMATCH: A heuristic algorithm for $B = 2^k$

Input $T = (V, E, r)$ with the root r and F leaves; each requested entry corresponds to a leaf node with a zero weight.

Output $\Pi_{2^k} = \{R_1, \dots, R_K\}$ where $|R_i| \leq B$

- 1: **for** level = 1 **to** k **do**
 - 2: $M \leftarrow \text{MATCH}(T)$
 - 3: **for** each pair $(i, j) \in M$ remove the leaf node j from T , and mark the leaf node i as representative
 - 4: Clean up the tree T so that all leaf nodes correspond to some requested entry
 - 5: Each remaining leaf node i corresponds to a part R_i where the nodes that are represented by i are put in R_i
-

diagonal entries and defer the discussion for the off-diagonal case until Section 4. We first give some relevant definitions.

3.4.1 Hypergraphs and the hypergraph partitioning problem

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices \mathcal{V} and a set of nets \mathcal{N} . Every net is a subset of vertices. Weights can be associated with vertices. We use $w(j)$ to denote the weight of the vertex v_j . Costs can be associated with nets. We use $c(h_i)$ to denote the cost associated with the net h_i .

$\Pi = \{\mathcal{V}_1, \dots, \mathcal{V}_K\}$ is a K -way vertex partition of $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ if each part is nonempty, parts are pairwise disjoint, and the union of the parts gives \mathcal{V} . In Π , a net is said to connect a part if it has at least one vertex in that part. The *connectivity set* $\Lambda(i)$ of a net h_i is the set of parts connected by h_i . The *connectivity* $\lambda(i) = |\Lambda(i)|$ of a net h_i is the number of parts connected by h_i . In Π , the weight of a part is the sum of the weights of vertices in that part.

In the hypergraph partitioning problem, the objective is to minimize

$$\text{cutsize}(\Pi) = \sum_{h_i \in \mathcal{N}} (\lambda(i) - 1) \times c(h_i). \quad (3.2)$$

This objective function is widely used in the VLSI community (Lengauer 1990) and in the scientific computing community (Aykanat, Pinar and Çatalyürek 2004, Çatalyürek and Aykanat 1999a, Uçar and Aykanat 2004); it is referred to as the *connectivity-1* cutsizes metric. The partitioning constraint is to satisfy a balancing constraint on part weights:

$$\frac{W_{max} - W_{avg}}{W_{avg}} \leq \varepsilon.$$

Here W_{max} is the largest part weight, W_{avg} is the average part weight, and ε is a predetermined imbalance ratio. This problem is NP-hard (Lengauer 1990).

3.4.2 The model

We build a hypergraph whose partition according to the cutsizes (3.2) corresponds to the total size of the factors loaded. Clearly, the requested entries (which correspond to the leaf nodes) are going to be the vertices of the hypergraph, so that a vertex partition will define a partition on

the requested entries. The more intricate part of the model is the definition of the nets. The nets correspond to edge disjoint paths in the tree, starting from a given node (not necessarily a leaf) and going up to one of its ancestors (not necessarily the root); each net is associated with a cost corresponding to the total size of the nodes in the corresponding path. We use $path(h)$ to denote the path (or the set of nodes of the tree) corresponding to a net h . A vertex i (corresponding to the leaf node i in the tree) will be in a net h if the solve for a_{ii}^{-1} passes through $path(h)$. In other words, if $path(h) \subset P(i)$, then $v_i \in h$. Therefore, if the vertices of a net h_n are partitioned among $\lambda(n)$ parts, then the factors corresponding to the nodes in $path(h_n)$ will have to be loaded $\lambda(n)$ times. As we load a factor at least once, the extra cost incurred by a partitioning is $\lambda(n) - 1$ for the net h_n . Given this observation, it is easy to see the equivalence between the total size of the loaded factors and the cutsizes of a partition plus the total weight of the tree.

We now define the hypergraph $\mathcal{H}_D = (\mathcal{V}_D, \mathcal{N}_D)$ for the diagonal case. Let $T = (V, E)$ be the tree corresponding to the modified elimination tree so that the requested entries correspond to the leaf nodes. Then, the vertex set \mathcal{V}_D corresponds to the leaf nodes in T . As we are interested in putting at most B many solves together, we assign a unit weight to the vertices of \mathcal{H} . The nets are best described informally. There is a net in \mathcal{N} for each internal node of T . The net h_n corresponding to the node n contains the set of vertices which correspond to the leaf nodes of subtree $T(n)$. The cost of h_n is equal to the weight of node n , i.e., $c(h_n) = w(n)$. This model can be simplified as follows: if a net h_n contains the same vertices as the net h_j where $j = parent(n)$, that is if the subtree rooted at node n and the subtree rooted at its father j has the same set of vertices, then the net h_j can be removed, and its cost can be added to the cost of the net h_n . This way the net h_n represents the path from node n to its parent j . After this transformation, we can also remove the nets with single vertices (these correspond to fathers of the leaf nodes with a single child), as these nets cannot contribute to the cutsizes. We note that the remaining nets will correspond to disjoint paths in the tree T and that their original node is the least common ancestor of two leaves.

Figure 3.3 shows an example of such a hypergraph: the requested entries are a_{11}^{-1} , a_{22}^{-1} , and a_{55}^{-1} . Therefore, $\mathcal{V} = \{1, 2, 5\}$ and $\mathcal{N} = \{h_1, h_2, h_4, h_5\}$ (net h_3 is removed according to the rule described above and the cost of h_2 includes the weight of nodes 2 and 3). Each net contains the leaf vertices which belong to the subtree rooted at its associated node, therefore : $h_1 = \{1\}, h_2 = \{2\}, h_4 = \{1, 2\}, h_5 = \{1, 2, 5\}$. Given, for example, the partition $V_1 = \{2\}$ and $V_2 = \{1, 5\}$ shown on the left of the figure, the cutsize is:

$$\begin{aligned} cutsize(V_1, V_2) &= c(h_1) \times (\lambda(h_1) - 1) + c(h_2) \times (\lambda(h_2) - 1) \\ &\quad + c(h_4) \times (\lambda(h_4) - 1) + c(h_5) \times (\lambda(h_5) - 1) \\ &= c(h_4) \times (2 - 1) + c(h_5) \times (2 - 1) \\ &= c(h_4) + c(h_5) . \end{aligned}$$

Consider the first part $V_1 = \{2\}$. We have to load the factors associated with the nodes 2, 3, 4, 5. Consider now the second part $V_2 = \{1, 5\}$. For this part, we have to load the factors associated with the nodes 1, 4, 5. Hence, the factors associated with the nodes 4 and 5 are loaded twice, while the factors associated with all other (internal) nodes are loaded only once. Since we have to access each node at least once, the extra cost due to the given partition is $w(4) + w(5)$, which is equal to the cutsize $c(h_4) + c(h_5)$.

The model encodes the partitioning problem exactly, in the sense that the cutsizes of a partition

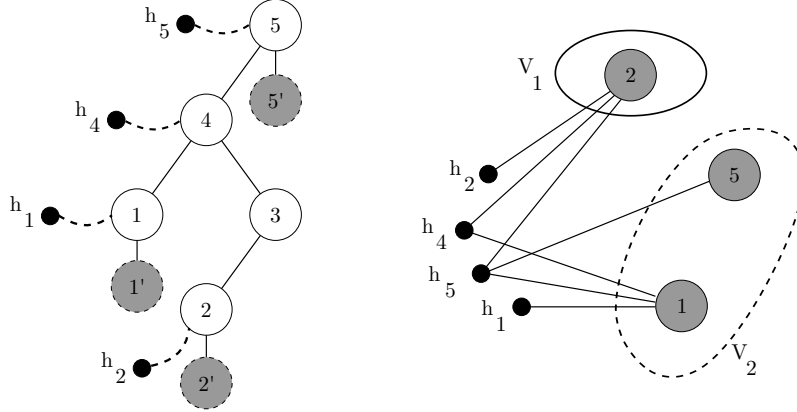


Figure 3.3: The entries a_{11}^{-1} , a_{22}^{-1} , and a_{55}^{-1} are requested. For each requested entry, a leaf node is added to the elimination tree as shown on the left. The hypergraph model for the requested entries is build as shown on the right.

is equal to the overhead incurred due to that partition; however, it can lead to huge amount of data and computation. Therefore, we envisage the use of this model only for the cases where a small set of entries is requested. But we believe that one can devise special data structures to hold the resulting hypergraphs, as well as specialized algorithms to partition them.

4 The off-diagonal case

As discussed before, the formulation for the diagonal case carries over to the off-diagonal case as well. But an added difficulty in this case is related to the actual implementation of the solver. Assume that we have to solve for a_{ij}^{-1} and a_{kj}^{-1} , that is, two entries in the same column of A^{-1} . As seen from the formula (1.1), reproduced below for convenience,

$$\begin{cases} y = L^{-1}e_j \\ a_{ij}^{-1} = (U^{-1}y)_i \end{cases}$$

only one y vector suffices. Similarly one can solve for the common nonzero entries in $U^{-1}y$ only once for i and k . This means that, for the forward solves with L , we can perform only one solve, and for the backward solves, we can solve for the variables in the path from the root to $lca(i, k)$ only once. Clearly, this will reduce the operation count. In an out-of-core context, we will load the same factors whether or not we keep the same number of right-hand sides throughout the computation (both in the forward and backward substitution phases). Avoiding the unnecessary repeated solves would only affect the operation count.

If we were to exclude the case when more than one entry in the same column of A^{-1} is requested, then we can immediately generalize our results and models developed for the case of diagonal entries to the off-diagonal case. Of course, the partitioning problem will remain NP-complete (it contains the instances with diagonal entries as a particular case). The lower bound can also be generalized to yield a lower bound for the case with arbitrary entries. Indeed, we only have to apply the same reasoning twice: one for the column indices of the requested entries, and one for the row indices of the requested entries. We can extend the model to cover the

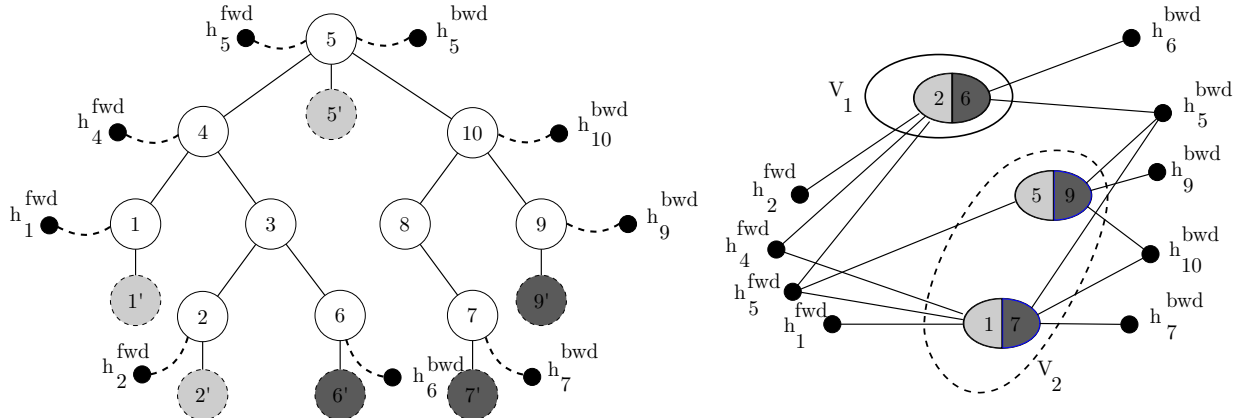


Figure 4.1: Example of hypergraph model for the general case: a_{71}^{-1} , a_{62}^{-1} and a_{95}^{-1} are requested.

case of multiple entries by duplicating nodes although this does result in solving for multiple vectors instead of a potential single solve. We extend the model by noting that when indices are repeated (say a_{ij}^{-1} and a_{kj}^{-1} are requested), we can distinguish them by assigning each occurrence to a different leaf node (we add two zero-weighted leaf nodes to the node j of the elimination tree). Then adding these two lower bounds yields a lower bound for the general case. However, in our experience, we have found this lower bound to be loose. Note that applying this lower bound to the case where only diagonal entries are requested yields the lower bound given in Theorem 3.1.

The POPART and the BISEMATCH heuristics do not naturally generalize to the off-diagonal case, because the generalized problem has a more sophisticated underlying structure. However the hypergraph partitioning-based approach works for arbitrary entries of the inverse. The idea is to model the forward and backward solves with two different hypergraphs, and then to partition these two hypergraphs simultaneously. It has been shown how to partition two hypergraphs simultaneously in Uçar and Aykanat (2004). The essential idea, which is refined in Uçar and Aykanat (2007), is to build a composite hypergraph by amalgamating the relevant vertices of the two hypergraphs, while keeping the nets intact. In our case, the two hypergraphs would be the model for the diagonal entries associated with the column subscripts (forward phase), and the model for the diagonal entries associated with the row subscripts (backward phase), again assuming that the same indices are distinguished by associating them with different leaf nodes. We have then to amalgamate any two vertices i and j where the entry a_{ij}^{-1} is requested.

Figure 4.1 shows an example where the requested entries are a_{71}^{-1} , a_{62}^{-1} and a_{95}^{-1} . The transformed elimination tree and the nets of the hypergraphs associated with the forward (h^{fwd}) and backward (h^{bwd}) solves are shown. Note that the nets h_3^{fwd} as well as h_3^{bwd} , h_4^{bwd} , and h_8^{bwd} are removed. The nodes of the tree which correspond to the vertices of the hypergraph for the forward solves are shaded with light grey; those nodes which correspond to the vertices of the hypergraph for the backward solves are shaded with dark grey. The composite hypergraph is shown in the right-hand figure. The amalgamation of light and dark grey vertices is done according to the requested entries (vertex i and vertex j are amalgamated for a requested entry a_{ij}^{-1}). A partition is given in the right-hand figure: $\Pi = \{\{a_{62}^{-1}\}, \{a_{71}^{-1}, a_{95}^{-1}\}\}$. The cut size is $c(h_5^{\text{bwd}}) + c(h_4^{\text{fwd}}) + c(h_5^{\text{fwd}})$. Consider the computation of a_{62}^{-1} . We need to load the L factors

associated with the nodes 2, 3, 4, and 5 and the U factors associated with 5, 4, 3 and 6. Now consider the computation of a_{71}^{-1} and a_{95}^{-1} ; the L factors associated with 1, 4, and 5, and the U factors associated with 5, 10, 8, 7, and 9 are loaded. In the forward solution, the L factors associated with 4 and 5 are loaded twice (instead of once if we were able to solve for all of them in a single pass), and in the backward solution the U factor associated with 5 is loaded twice (instead of once). The cutsize again corresponds to these extra loads.

We note that building such a hypergraph for the case where only diagonal entries are requested yields the hypergraph of the previous section, where each hyperedge is repeated twice.

5 Experiments

We conduct three sets of experiments. In the first set, we compare the quality of the results obtained by the POPART and BISEMATCH heuristics using Matlab implementations of these algorithms. For these experiments, we created a large set of TREEPARTITIONING problems, each of which is associated with computing some diagonal entries in the inverse of a sparse matrix. In the second set of experiments, we use an implementation of POPART in Fortran that we integrated into the MUMPS solver (Amestoy et al. 2001). We use this to investigate the performance of POPART on practical cases using the out-of-core option of MUMPS. In this set of experiments, we compute a set of entries from the diagonal of the inverse of matrices from two different data sets; the first set contains a few matrices coming from the astrophysics application (Bouchet et al. 2005) briefly described in Section 1; the second set contains some more matrices that are publicly available. In the third set of experiments, we carry out some experiments with the hypergraph model for the off-diagonal case.

5.1 Assessing the heuristics

Our first set of experiments compares the heuristics POPART and the BISEMATCH which were discussed in Sections 3.1 and 3.3. We have implemented these heuristics in Matlab. We use a set of matrices from the University of Florida (UFL) sparse matrix collection (<http://www.cise.ufl.edu/research/sparse/matrices/>). The matrices we choose satisfy the following characteristics: $10000 \leq N \leq 100000$, the average number of nonzeros per row is greater than or equal to 2.5, and in the UFL index the `posdef` field is set to 1. At the time of writing, there were a total of 61 matrices satisfying these properties. We have ordered the matrices using the `metisnd` routine of the Mesh Partitioning Toolbox (Gilbert, Miller and Teng 1998), and built the elimination tree associated with the ordered matrices using the `etree` function of Matlab. We have experimented with block sizes $B \in \{2, 4, 8, 16, 32, 64, 128, 256\}$. We have assigned random weights in the range 1–200 to tree nodes. Then, for each $P \in \{0.05, 0.10, 0.20, 0.40, 0.60, 0.80, 1.00\}$, we have created 10 instances (except for $P = 1.00$) by randomly selecting $P \times N$ integers between 1 and N and designating them as the requested entries in the diagonal of the inverse. Notice that for a given triplet of a matrix, B , and P , we have 10 different trees to partition, resulting in a total of $10 \times 6 \times 8 \times 61 + 8 \times 61 = 29768$ TREEPARTITIONING problems.

We summarize the results in Table 5.1 by giving results with $B \in \{4, 16, 64, 256\}$ (the last row relates to all B values mentioned in the previous paragraph). Our subsequent discussion relates to the complete set of experiments and not just those shown in Table 5.1. In order to create this table, we computed the lower bound for all tree partitioning instances, and computed

the ratio of the costs found by POPART and BISEMATCH to the lower bound. Next, for a triplet of a matrix, B , and P , we took the average of the ratios of the 10 random instances, and stored that average result for the triplet. Then we took the minimum, the maximum, and the average of the 61 different triplets with the same B and P . As seen in this table, both heuristics obtain results that are close to the lower bound: POPART’s average result is about 1.04 times the lower bound, and BISEMATCH’s average result is about 1.01 times the lower bound. The POPART heuristic attains the exact lower bound in only one triplet, while BISEMATCH attains the lower bound for all instances with $B = 2$ (recall that it is based on the exact algorithm MATCH) and for some other five triplets. The maximum deviation from the lower bound is about 10% with BISEMATCH, whereas it is 30% with POPART. Given that, in most cases, the algorithms perform close to the average figures, we conclude that both are efficient enough to be useful in the context of the out-of-core solver.

The BISEMATCH heuristic almost always obtains better results than POPART: in only 7 out of 56×61 triplets, did POPART obtain better results than BISEMATCH. For all P , the performance of BISEMATCH with respect to the lower bound became worse as B increases. Although there are fluctuations in the performance of POPART for small values of P , e.g., for 0.05 and 0.10, for larger values of P , the performance also becomes worse with larger values of B . We suspect that the lower bound might be loose for large values of B . For all B , the performance of BISEMATCH with respect to the lower bound improves when P increases. A similar trend is observable for POPART, except for a small deviation for $B = 256$. Recall that the trees we use here come from a nested dissection ordering. Such ordering schemes are known to produce wide and balanced trees. This fact, combined with the fact that when a high percentage of the diagonal entries are requested, the trees will not have their structure changed much by removal and addition of leaf nodes, may explain why the heuristics perform better at larger values of P for a given B .

5.2 Practical tests with a direct solver

We have implemented the heuristic POPART in Fortran, and integrated it into the MUMPS solver (Amestoy et al. 2001). The implementation of the computation of a set of inverse entries exploiting sparsity within MUMPS is described in Slavova (2009). In this section, we give results obtained by using MUMPS with the out-of-core option, and a nested dissection ordering provided by MeTiS (Karypis and Kumar 1998). All experiments have been performed with direct I/O access to files so that we can guarantee effective disk access independently of both the size of the factors and the size of the main memory. The benefits resulting from the use of direct I/O mechanisms during the solution phase are discussed in Amestoy, Duff, Guermouche and Slavova (2010). All results are obtained on a dual-core Intel Core2 Duo P8800 processor having a 2.80 GHz clock speed. We have used only one of the cores, and we did not use threaded BLAS. We use four matrices from the real life astrophysics application (Bouchet et al. 2005) briefly described in Section 1. The names of these matrices start with CESR and continue with the size of the matrix. We use an additional set of four matrices (af23560, ecl32, stokes64, boyd1) from the UFL sparse matrix collection with very different nonzero patterns, yielding a set of elimination trees with varying structural properties (such as height and width of the tree, and variations in node degrees).

Table 5.2 shows the total size of the factors loaded and the execution time of the solution phase of MUMPS with different settings and partitions. All diagonal entries of the inverse of

Table 5.1: The performance of the proposed POPART and BISEMATCH heuristics with respect to the lower bound. The numbers represent the average over 61 different matrices of the ratios of the results of the heuristics to the lower bound discussed in the text. The column B corresponds to the maximum allowable block size; the column P corresponds to the requested percentage of diagonal entries.

B	P	POPART			BISEMATCH		
		min	max	avg	min	max	avg
4	0.05	1.0011	1.1751	1.0278	1.0000	1.0139	1.0013
	0.10	1.0005	1.1494	1.0192	1.0000	1.0073	1.0005
	0.20	1.0003	1.0945	1.0119	1.0000	1.0052	1.0003
	0.40	1.0001	1.0585	1.0072	1.0000	1.0031	1.0001
	0.60	1.0001	1.0449	1.0053	1.0000	1.0019	1.0001
	0.80	1.0000	1.0367	1.0043	1.0000	1.0029	1.0001
	1.00	1.0000	1.0491	1.0038	1.0000	1.0101	1.0002
16	0.05	1.0050	1.1615	1.0592	1.0000	1.0482	1.0113
	0.10	1.0026	1.1780	1.0485	1.0000	1.0553	1.0075
	0.20	1.0016	1.2748	1.0374	1.0000	1.0334	1.0035
	0.40	1.0007	1.1898	1.0246	1.0000	1.0230	1.0016
	0.60	1.0005	1.1431	1.0186	1.0000	1.0166	1.0010
	0.80	1.0004	1.1136	1.0154	1.0000	1.0190	1.0011
	1.00	1.0003	1.1052	1.0133	1.0000	1.0096	1.0008
64	0.05	1.0132	1.1581	1.0800	1.0000	1.0797	1.0275
	0.10	1.0101	1.1691	1.0715	1.0002	1.0584	1.0196
	0.20	1.0054	1.1389	1.0599	1.0001	1.0506	1.0125
	0.40	1.0030	1.1843	1.0497	1.0000	1.0437	1.0079
	0.60	1.0020	1.2362	1.0407	1.0000	1.1022	1.0072
	0.80	1.0015	1.3018	1.0383	1.0000	1.0344	1.0044
	1.00	1.0014	1.2087	1.0315	1.0000	1.0141	1.0024
256	0.05	1.0050	1.1280	1.0651	1.0000	1.0867	1.0342
	0.10	1.0127	1.1533	1.0721	1.0003	1.0911	1.0314
	0.20	1.0133	1.1753	1.0730	1.0002	1.0722	1.0257
	0.40	1.0093	1.1598	1.0668	1.0003	1.0540	1.0187
	0.60	1.0068	1.1621	1.0602	1.0002	1.0572	1.0174
	0.80	1.0068	1.1314	1.0563	1.0001	1.0515	1.0120
	1.00	1.0043	1.1203	1.0495	1.0001	1.0677	1.0118
Over all triplets		1.0000	1.3018	1.0359	1.0000	1.1110	1.0079

the given matrices are computed with $B = \{16, 64\}$. In this table, the values in column “Lower bound” are computed according to Theorem 3.1. The column “NoES” corresponds to the computational scheme where the sparsity of the right-hand side vectors involved in computing the diagonal entries is not exploited. The columns “ES-Nat” and “ES-POP” correspond to the computational scheme where the sparsity of the right-hand side vectors are exploited to speed up the solution process. These columns correspond, respectively, to the natural partitioning (the indices are partitioned in the natural order into blocks of size B) and to the POPART heuristic. As seen in column “ES-Nat”, most of the gain, in the total size of the factors loaded and in the execution time, is due to exploiting the sparsity of the right-hand side vectors. Furthermore, when reordering the right-hand sides following a postorder (POPART), the total number of loaded factors is once again reduced significantly, resulting in a noticeable impact on the execution time. We also see that the larger the block size, the better POPART performs compared to the natural partitioning. This can be intuitively explained as follows: within an epoch, computations are performed on a union of paths, hence the natural ordering is likely to have more trouble with increasing epoch size, because it will combine nodes far from each other.

We see that the execution times are proportional to the total volume of loaded factors. On a majority of problems, the execution time is largely dominated by the time spent reading the factors from the disk, which explains this behaviour: for example, on matrix ecl32, 95% of the time is spent on I/O. On a few problems only, the time spent in I/O represents a significant but not dominant part of the runtime, hence slightly different results (e.g., on CESR72358, the time for loading the factors represents less than a third of the total time for ES-POP). On such matrices, increasing the block size is likely to increase the number of operations and thus the time, as explained in Section 6.1 for the in-core case.

5.3 Hypergraph model

We have performed a set of experiments with the hypergraph model introduced in Section 4, in an attempt to see how it performs in practice, and to set a base case method for future developments. Table 5.3 summarizes some tests with the model. The matrices are the same as before. The tests are again conducted using the out-of-core option of MUMPS, and standard settings including an ordering based on nested dissection. A random selection of $N/10$ off-diagonal entries (no two in the same column) are computed with $B = \{16, 64\}$. The table displays the lower bound (given in Section 4), and the total size of the factors loaded with a POPART partition on the column indices, and with a partition on the hypergraph models using PaToH (Çatalyürek and Aykanat 1999b) with default options, except that we have requested a tighter balance among part sizes. As expected, the formulation based on hypergraph partitioning obtains better results than that based on post-order. There is, however, a huge difference between the lower bounds and the performance of the heuristics. We performed additional tests with the hypergraph model on the diagonal case and observed that its performance was similar to that of POPART (which we have shown to be very effective). We think therefore that the performance of the hypergraph based formulation should be again reasonable and that the lower bound is too loose to judge the effectiveness of the heuristic. However, as pointed out before, hypergraph models can rapidly become huge, so further studies are needed.

Table 5.2: The total size of the loaded factors and execution times with MUMPS with two different computational schemes. All diagonal entries are requested. The out-of-core executions use direct I/O access to the files. NoES refers to the traditional solution without exploiting sparsity. The columns ES-Nat and ES-POP refer to exploiting the sparsity of the right-hand side vectors under, respectively, a natural partitioning and the POPART heuristic.

matrix	B	Lower bound	Total size of the loaded factors (MBytes)			Running time of the solution phase (s.)		
			NoES	ES-Nat	ES-POP	NoES	ES-Nat	ES-POP
CESR21532	16	5403	63313	7855	5422	1113.8	69.4	39.3
	64	1371	15828	2596	1389	359.6	38.1	16.6
CESR46799	16	2399	114051	3158	2417	3962.3	76.7	47.7
	64	620	28512	1176	635	866.3	51.3	28.5
CESR72358	16	1967	375737	6056	2008	10800.9	263.7	71.8
	64	528	93934	4796	571	3174.0	274.1	52.0
CESR148286	16	8068	1595645	16595	8156	43396.7	720.7	268.5
	64	2092	398911	11004	2179	14049.3	726.7	199.8
af2356	16	16720	114672	17864	16745	2080.6	241.1	197.6
	64	4215	28668	5245	4245	668.5	121.0	59.5
ecl32	16	95478	618606	141533	95566	12184.7	2726.3	1760.6
	64	23943	154651	43429	24046	3525.5	974.1	482.9
stokes64	16	721	8503	1026	726	131.2	14.2	8.5
	64	185	2125	425	189	48.8	10.2	4.1
boyd1	16	2028	75521	4232	2031	16551.2	389.8	214.9
	64	515	18880	1406	518	5492.7	230.5	121.2

Table 5.3: The total size of the loaded factors and execution times with MUMPS with two different computational schemes. A random set of $N/10$ off-diagonal entries are computed with MUMPS. Out-Of-Core executions are with direct I/O access to the files. The columns ES-POp and ES-HP correspond, respectively, to the case where a post-order on the column indices and a hypergraph partitioning routine are used to partition the requested entries into blocks of size 16 and 64.

matrix	B	Lower bound	Total size of the loaded factors (MBytes)		Running time of the solution phase (s.)	
			ES-POp	ES-HP	ES-POp	ES-HP
CESR21532	16	563	1782	999	56.6	16.9
	64	164	703	464	28.3	13.6
CESR46799	16	264	549	416	32.2	25.1
	64	93	232	195	35.5	25.3
CESR72358	16	242	1124	868	122.9	73.8
	64	116	794	598	99.5	72.8
CESR148286	16	905	3175	2693	426.0	321.7
	64	345	2080	1669	281.0	235.8
af23560	16	1703	3579	2463	109.2	66.4
	64	458	1219	1003	47.1	34.3
ecl32	16	9617	22514	12615	507.7	265.2
	64	2483	7309	4664	199.2	119.9
stokes64	16	77	188	149	2.9	2.3
	64	26	75	74	1.9	1.7
boyd1	16	205	481	258	39.0	34.4
	64	55	198	93	25.9	24.2

6 Similar problems and related work

In this section, we briefly present extensions and variations of the problem. Firstly, we show the difference between the in-core and the out-of-core cases. Then we address the problem of exploiting parallelism: indeed the partitionings presented above can limit tree-parallelism, and we propose a simple heuristic to remedy this.

6.1 In-core case

In an in-core context, the relevant metric is the number of floating-point operations (*flops*) performed. When processing several right-hand sides at the same time, block computations are performed on the union of the structures of these vectors, hence there are more computations than there would be if these right-hand sides were processed one-by-one; of course, the interest is to benefit from dense kernels, such as the BLAS, and thus to process the right-hand sides by blocks of reasonable size.

A first observation can be made about the block size: in the in-core context, the optimal block size, in terms of the number of floating-point operations is one, because no extra operations are performed. Conversely, putting all the right-hand sides in a single block represents the worst case, because a maximum amount of extra operations is introduced. In the out-of-core case, things are completely different: processing all the right-hand sides in one shot is the best strategy, because all the nodes (pieces of factors stored on the hard drive) are loaded only once; conversely, processing all the right-hand sides one by one implies accessing each node a maximum number of times. Therefore, the choice of a block size will be different in each case: for in-core, we will choose a block size which gives a good trade-off between dense kernel efficiency and the number of extra operations introduced; for out-of-core, we will try to maximize the block size (constrained by the available memory).

One might think that, for a given block size, partitions that perform well in out-of-core should be efficient in the in-core case as well. Unfortunately, this is not the case, and Figure 6.1 provides a counter-example. The tree shown corresponds to the usual assembly tree. Assume that each internal node has unit weight and that a unit number of operations is performed for each requested entry in a given set. The partition $\{\{1, 2\}, \{3\}\}$ is better than the partition $\{\{1, 3\}, \{2\}\}$ in the out-of-core context, but in the in-core case, the second partition results in fewer operations.

In a different context (Yamazaki, Li and Ng 2010), a post-order based method and a hypergraph partitioning model have been shown to be useful in reducing the number of (redundant) operations in supernodal triangular solves with many sparse right-hand sides. We have not investigated the effects on the computation of the entries of the inverse; however, as outlined above, the objectives in the out-of-core case, for which our heuristics are proposed, are different than the objective of minimizing the operation count.

6.2 Parallelism

In MUMPS, a subtree to subcube mapping (Geist and Ng 1989) is performed on the lower part of the tree during the analysis phase. Nodes in the lower part of the tree are likely to be mapped on a single processor, whereas nodes in the upper part of the tree are mapped onto several processors. Since the partitioning strategies described above tend to put together nodes which are close in the elimination tree, few processors (probably only one) will be active in the lower part of tree

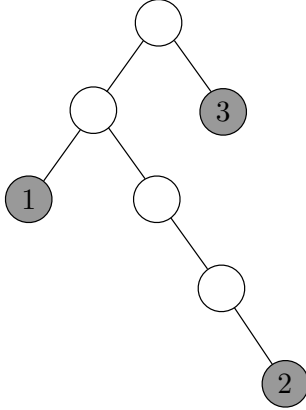


Figure 6.1: Each internal node has unit weight, and the number of operations performed at each node is equal to the number of entries in the processed block. In the out-of-core context, the partition $\{\{1, 2\}, \{3\}\}$ is better than the partition $\{\{1, 3\}, \{2\}\}$ because the total size of loaded factors are 5 units (4 for the first block and 1 for the second) vs 6 units ($2 + 4$); however, in the in-core context, the situation is reversed because the total operation counts are 10 (2 operations for 4 nodes and 1 operation for 2 nodes) vs 8 (decomposed as $2 \times 2 + 1 \times 4$).

when processing a block of right-hand sides. An *interleaving* strategy was suggested in Slavova (2009) to remedy this situation: it consists in interleaving the different blocks of right-hand sides so that every processor will be active when a block of entries is computed. The main drawback of this strategy is that it tends to lose the overlapping gained by partitioning. When local disks are attached to processors, then the gain in global bandwidth balances this loss. Further work is needed to design strategies which partition the requested entries so that parallelism is ensured, without increasing either the number of operations or the number of disk accesses.

6.3 Related work

Most of the related work addresses the case of computing the whole diagonal of the inverse of a given matrix. Among these, in the studies regarding the applications mentioned in Section 1, the diagonal entries are computed using direct methods. Tang and Saad (2009) address the same problem (computing the diagonal of the inverse) with an iterative method focusing on matrices whose inverses have a decay property.

For the general off-diagonal case, not much work has been done. To compute entries in A^{-1} , one can use the algorithm given in Erisman and Tinney (1975). This algorithm relies on equations derived by Takahashi et al. (1973). Given an LU factorization of an $N \times N$ sparse matrix A , the algorithm computes the parts of the inverse of A that correspond to the positions of the nonzeros of $(L + U)^T$, starting from entry (N, N) and proceeding in a reverse Crout order. At every step, an entry of the inverse is computed using the factors L and U and the already computed entries of the inverse. This approach is later extended (Niessner and Reichert 1983) for a set of entries of the inverse, rather than the whole set in the pattern of $(L + U)^T$. The algorithm has been implemented in a multifrontal-like approach (Campbell and Davis 1995).

If all entries in the pattern of $(L + U)^T$ are requested, then the method that implements the algorithm in Erisman and Tinney (1975) might be advantageous, whereas methods based

on the traditional solution of linear systems have to solve at least n linear systems and require considerably more memory. On the other hand, if a set of entries in the inverse is requested, any implementation based on the equations by Takahashi et al. should set up necessary data structures and determine the computational order to compute all the entries that are necessary to compute those requested. This seems to be a rather time consuming operation.

7 Conclusion

We have addressed the problem of efficiently computing some entries of the inverse of a sparse matrix, from computed sparse factors and using the elimination tree. We have shown that only factors from paths between a node and the root are required both in the forward and the backward substitution phases. We then examined the efficient computation of multiple entries of the inverse, particularly in the case where the factors are held out-of-core.

We have proposed several strategies for minimizing the cost of computing multiple inverse entries. The issue here is that memory considerations restrict how many entries can be computed simultaneously so that we need to partition the requested entries to respect this constraint. We showed that this problem is NP-complete so that it is necessary to develop heuristics for doing this partitioning.

We describe a very simple heuristic, POPART, which is based on a post-ordering of the elimination tree and then a partitioning of the nodes in sequential parts according to this post-order. We showed that this is a 2-approximation algorithm. Although we showed that the TREEPARTITIONING problem cannot be approximated arbitrarily closely, there remains a gap to fill, and in future work we will strive at designing approximation algorithms with a better ratio.

We presented an exact algorithm for the case when two entries are computed at a time. By using this exact algorithm repeatedly, we developed another heuristic, BISEMATCH, for partition sizes that are powers of two. We performed extensive tests on the heuristics, and have concluded that both POPART and BISEMATCH perform very well on average, where the worst case the performance of the BISEMATCH is better. By comparing the performance of these heuristics with computable lower bounds we saw that they give very effective partitionings. We implemented the POPART heuristic within the MUMPS solver, and reported experimental results with MUMPS. These confirmed the effectiveness of the POPART heuristic.

The heuristics POPART and BISEMATCH were designed for the case where only diagonal entries of the inverse are requested. To accommodate the case when off-diagonal entries are wanted, we have proposed a formulation based on a hypergraph partitioning. In this model, a hypergraph is built so that the cutsize of the partition corresponds exactly to the increase in the total size of factors loaded. Although the size of the hypergraph model can be large, the model is powerful enough to represent both the diagonal and the off-diagonal cases. We also performed tests with the hypergraph model, and concluded that it can be used effectively for cases where a small number of entries in the inverse are requested.

We briefly described a technique to improve the performance for parallel execution, and showed differences that apply when the factorization is held in-core. Although we have made the first steps for showing the efficient computation of off-diagonal inverse entries more work should be done in that case to obtain practical algorithms when many entries are requested.

References

- Agullo, E. (2008), On the out-of-core factorization of large sparse matrices, PhD thesis, Ecole Normale Supérieure de Lyon.
- Amestoy, P. R., Duff, I. S., Guermouche, A. and Slavova, Tz. (2010), ‘Analysis of the solution phase of a parallel multifrontal approach’, *Parallel Computing* **36**, 3–15. doi:10.1016/j.parco.2009.06.001.
- Amestoy, P. R., Duff, I. S., Koster, J. and L’Excellent, J.-Y. (2001), ‘A fully asynchronous multifrontal solver using distributed dynamic scheduling’, *SIAM Journal on Matrix Analysis and Applications* **23**(1), 15–41.
- Amestoy, P. R., Guermouche, A., L’Excellent, J.-Y. and Pralet, S. (2006), ‘Hybrid scheduling for the parallel solution of linear systems’, *Parallel Computing* **32**(2), 136–156.
- Aykanat, C., Pinar, A. and Çatalyürek, Ü. V. (2004), ‘Permuting sparse rectangular matrices into block-diagonal form’, *SIAM Journal on Scientific Computing* **25**(6), 1860–1879.
- Björck, Å. (1996), *Numerical methods for least squares problems*, Society for Industrial Mathematics.
- Bouchet, L., Roques, J.-P., Mandrou, P., Strong, A., Diehl, R., Lebrun, F. and Terrier, R. (2005), ‘INTEGRAL SPI observation of the galactic central radian: Contribution of discrete sources and implication for the diffuse emission 1’, *The Astrophysical Journal* **635**(2), 1103–1115.
- Campbell, Y. E. and Davis, T. A. (1995), Computing the sparse inverse subset: an inverse multifrontal approach, Technical Report TR-95-021, CIS Dept., Univ. of Florida.
- Cauley, S., Jain, J., Koh, C. K. and Balakrishnan, V. (2007), ‘A scalable distributed method for quantum-scale device simulation’, *Journal of Applied Physics* **101**, 123715.
- Çatalyürek, Ü. V. and Aykanat, C. (1999a), ‘Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication’, *IEEE Transactions on Parallel and Distributed Systems* **10**(7), 673–693.
- Çatalyürek, Ü. V. and Aykanat, C. (1999b), *PaToH: A Multilevel Hypergraph Partitioning Tool, Version 3.0*, Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH is available at <http://bmi.osu.edu/~umit/software.htm>.
- Duff, I. S., Erisman, A. M., Gear, C. W. and Reid, J. K. (1988), ‘Sparsity structure and Gaussian elimination’, *SIGNUM Newsletter* **23**(2), 2–8.
- Erisman, A. M. and Tinney, W. F. (1975), ‘On computing certain elements of the inverse of a sparse matrix’, *Comm. ACM* **18**, 177–179.
- Garey, M. R. and Johnson, D. S. (1979), *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA.
- Geist, G. A. and Ng, E. G. (1989), ‘Task scheduling for parallel sparse Cholesky factorization’, *International Journal of Parallel Programming* **18**(4), 291–314.

- Gilbert, J. R. and Liu, J. W. H. (1993), ‘Elimination structures for unsymmetric sparse LU factors’, *SIAM J. Matrix Analysis and Applications*.
- Gilbert, J. R., Miller, G. L. and Teng, S.-H. (1998), ‘Geometric mesh partitioning: Implementation and experiments’, *SIAM Journal on Scientific Computing* **19**(6), 2091–2110.
- Karypis, G. and Kumar, V. (1998), *MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*, University of Minnesota, Department of Comp. Sci. and Eng., Army HPC Research Center, Minneapolis.
- Lengauer, T. (1990), *Combinatorial algorithms for integrated circuit layout*, John Wiley & Sons, Inc. New York, NY, USA.
- Lin, L., Lu, J., Ying, L., Car, R. and E, W. (2009), ‘Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems’, *Communications in Mathematical Sciences* **7**(3), 755–777.
- Liu, J. W. H. (1990), ‘The role of elimination trees in sparse factorization’, *SIAM Journal on Matrix Analysis and Applications* **11**(1), 134–172.
- Luisier, M., Schenk, A., Fichtner, W. and Klimeck, G. (2006), ‘Atomistic simulation of nanowires in the $sp^3d^5s^*$ tight-binding formalism: From boundary conditions to strain calculations’, *Physical Review B* **74**(20), 205323.
- Niessner, H. and Reichert, K. (1983), ‘On computing the inverse of a sparse matrix’, *International Journal for Numerical Methods in Engineering* **19**(10), 1513–1526.
- Slavova, Tz. (2009), Parallel triangular solution in an out-of-core multifrontal approach for solving large sparse linear systems, PhD thesis, Institut National Polytechnique de Toulouse, Toulouse, France.
- Takahashi, K., Fagan, J. and Chin, M. (1973), Formation of a sparse bus impedance matrix and its application to short circuit study, in ‘Proceedings 8th PICA Conference, Minneapolis, Minnesota’.
- Tang, J. and Saad, Y. (2009), A probing method for computing the diagonal of the matrix inverse, Technical Report umsi-2010-42, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, MN.
- Uçar, B. and Aykanat, C. (2004), ‘Encapsulating multiple communication-cost metrics in partitioning sparse rectangular matrices for parallel matrix-vector multiplies’, *SIAM Journal on Scientific Computing* **25**(6), 1837–1859.
- Uçar, B. and Aykanat, C. (2007), ‘Revisiting hypergraph models for sparse matrix partitioning’, *SIAM Review* **49**, 595–603.
- Yamazaki, I., Li, X. S. and Ng, E. G. (2010), ‘Partitioning, Load Balancing, and Matrix Ordering in a Parallel Hybrid Solver’, Presentation at SIAM Conference on Parallel Processing for Scientific Computing (PP10).