A GEOMETRIC APPROACH TO MATRIX ORDERING

B. O. FAGGINGER AUER* AND R. H. BISSELING†

Abstract. We present a recursive way to partition hypergraphs which creates and exploits hypergraph geometry and is suitable for many-core parallel architectures. Such partitionings are then used to bring sparse matrices in a recursive Bordered Block Diagonal form (for processor-oblivious parallel LU decomposition) or recursive Separated Block Diagonal form (for cache-oblivious sparse matrix–vector multiplication). We show that the quality of the obtained partitionings and orderings is competitive by comparing obtained fill-in for LU decomposition with SuperLU (with better results for 8 of the 28 test matrices) and comparing cut sizes for sparse matrix–vector multiplication with Mondriaan (with better results for 4 of the 12 test matrices). The main advantage of the new method is its speed: it is on average 21.6 times faster than Mondriaan.

Key words. hypergraphs, k-means, LU decomposition, nested dissection, partitioning, sparse matrices, visualization

AMS subject classifications. 05C65, 05C70, 65F05, 65F50, 65Y05

1. Introduction.

With the increased development and availability of many-core processors (both CPUs and GPUs) it is important to have algorithms that can make use of these architectures. To this end, we present a new recursive hypergraph partitioning algorithm that uses the underlying geometry of the hypergraph to generate the partitioning (which is largely done using shared-memory parallelism). Hypergraph geometry may either be provided from the problem at hand or generated by the partitioning software. This entire process is illustrated in Fig. 1.1.

1.1. Hypergraphs.

We will start with a brief introduction to hypergraphs (see [6] for more information) and the ways in which they can be related to sparse matrices.

DEFINITION 1.1. A *hypergraph* is a pair $G = (V, E)$ where V is a set, the *vertices* of the hypergraph G, and E a collection of subsets of V (so for all $e \in E$, $e \subseteq V$), the *hyperedges* or *nets* of G (see Fig. 1.2). We call a hypergraph $G = (V, E)$ *weighted* when it is paired with functions $w : V \to [0, \infty[$ and $c : E \to [0, \infty[$ which assign *weights* $w(v) \geq 0$ and *costs* $c(e) \geq 0$ to vertices $v \in V$ and hyperedges $e \in E$, respectively. We call a hypergraph $G = (V, E)$ simply a *graph* if all hyperedges $e \in E$ are of the form $e = \{v, w\}$ with $v, w \in V$; in this case the graph is undirected and the hyperedges are called *edges*. We call a hypergraph $G = (V, E)$ *finite* if V is a finite set, in which case $|E| \leq 2^{|V|}$, so E is finite as well.

Hypergraphs possess a natural notion of *duality*, where the roles played by the vertices and hyperedges are interchanged.

DEFINITION 1.2. Let $G = (V, E)$ be a hypergraph.

Then we define its *dual hypergraph* as $G^* := (V^*, E^*)$ where $V^* := E$ and

$$E^* := \{\{e \in E \mid e \ni v\} \mid v \in V\}.$$

If the hypergraph is weighted, we also exchange the vertex weights and hyperedge costs to make its dual a weighted hypergraph.

The dual of the dual of a hypergraph is isomorphic to the original hypergraph, $(G^*)^* \cong G$.

*Department of Mathematics, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, the Netherlands (B.O.FaggingerAuer@uu.nl).

†Department of Mathematics, Utrecht University, P.O. Box 80010, 3508 TA Utrecht, the Netherlands (R.H.Bisseling@uu.nl).

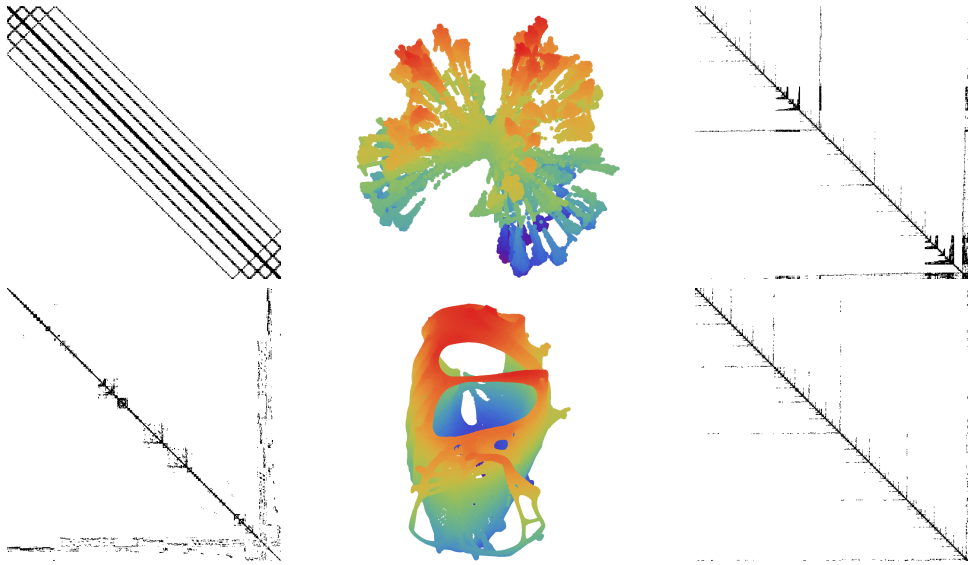

FIGURE 1.1. *The matrices* `twotone` *(top) and* `ford2` *(bottom). For the original matrix (left), a visual representation is created (middle), which in turn is used to permute the matrix to recursive Bordered Block Diagonal form (right).*

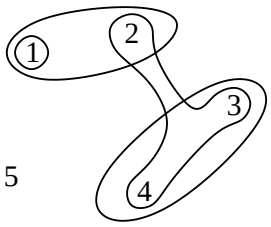

FIGURE 1.2. *Hypergraph* $G = (V, E)$ *with* $V = \{1, 2, 3, 4, 5\}$ *and* $E = \{\{1\}, \{1, 2\}, \{2, 3, 4\}, \{3, 4\}\}$.

One direct application of hypergraphs is to use them to represent sparse matrices, see Table 1.1. We can make a number of observations about these representations.

1. The symmetric representation is only sensible if the matrix is structurally symmetric, because only then we can recover the nonzero pattern of the original matrix from its representation.
2. The bipartite representation is a bipartite graph with the two parts consisting of the rows $(r_1, \ldots, r_m)$ and the columns $(c_1, \ldots, c_n)$ of the matrix.
3. The symmetric and bipartite representations are both undirected graphs instead of hypergraphs and the size of the symmetric representation is about half that of the bipartite representation.
4. The column-net and row-net representations are each other's dual.
5. The finegrain and bipartite representations are each other's dual. This is also reflected in the fact that the hyperedges of the finegrain representation can be partitioned into two disjoint sets (the row and column hyperedges).

We will make use of these observations in Section 4.

Name	V	E
Symmetric [30]	$\{1,\ldots,m\}$	$\{\{i,j\} \mid 1 \le i \le m, 1 \le j \le n, a_{i\,j} \neq 0\}$
Bipartite [21]	$\{r_1,\ldots,r_m,c_1,\ldots,c_n\}$	$\{\{r_i,c_j\} \mid 1 \le i \le m, 1 \le j \le n, a_{i\,j} \neq 0\}$
Column-net [7]	$\{r_1,\ldots,r_m\}$	$\{\{r_i \mid 1 \le i \le m, a_{i\,j} \neq 0\} \mid 1 \le j \le n\}$
Row-net [7]	$\{c_1,\ldots,c_n\}$	$\{\{c_j \mid 1 \le j \le n, a_{i\,j} \neq 0\} \mid 1 \le i \le m\}$
Finegrain [8]	$\{v_{i\,j} \mid a_{i\,j} \neq 0\}$	$\underbrace{\{\{v_{i\,j} \mid 1 \le i \le m, a_{i\,j} \neq 0\} \mid 1 \le j \le n\}}_{\text{column hyperedges}}$ $\cup \underbrace{\{\{v_{i\,j} \mid 1 \le j \le n, a_{i\,j} \neq 0\} \mid 1 \le i \le m\}}_{\text{row hyperedges}}$

TABLE 1.1
Several common representations of an $m \times n$-matrix $A = (a_{i\,j})$ by a hypergraph $G = (V, E)$.

1.2. Visual representations. As stated in Section 1, we would like to exploit the underlying geometry of a hypergraph, usually representing a sparse matrix.

DEFINITION 1.3. Let $G = (V, E)$ be a given hypergraph. Then a *visual representation* of G in $d \in \mathbf{N}$ dimensions is a mapping

$$V \to \mathbf{R}^d$$

that reflects the structure of the underlying hypergraph.

This definition is not very precise and therefore we will illustrate it by looking at a few examples. Sometimes the visual representation of a matrix is directly available, for instance if the matrix is based on a triangulated mesh, such as the following Example 1.4. In other cases, the visual representation can be generated from the problem that the matrix represents, see Example 1.5. If no such information is available at all, we generate the visual representation ourselves, as discussed in Section 2.

EXAMPLE 1.4. The `pothen` collection of matrices, available from [12], consists of NASA structural engineering matrices collected by A. Pothen. We will take a closer look at the square pattern[1] matrices from this collection, which have a natural visual representation. Each row/column of these matrices corresponds to a vertex (the coordinates of which are supplied in a separate file) and each nonzero to an edge between the vertices corresponding to the row and column to which the nonzero belongs. We can take a look at the matrices and their corresponding visual representation by plotting these vertices, as is done in Fig. 1.3. These vertices give a visual representation of the symmetric hypergraph representation of the sparse matrix.

EXAMPLE 1.5. Another important example of matrices with a natural visual representation are those arising in finite-element methods. In this example, we will consider the Laplace equation on a compact smooth d-dimensional closed submanifold $\Omega \subseteq \mathbf{R}^d$ with compact smooth boundary $\partial\Omega$. Let

$$V := \{f \in C^\infty(\Omega, \mathbf{R}) \mid f,\ \|\nabla f\| \in L^2(\Omega, \mathbf{R}),\ f|_{\partial\Omega} = 0\}$$

be the collection of all smooth real-valued functions on Ω that are square-integrable, have square-integrable derivative, and vanish on the boundary $\partial\Omega$. Here the inner product is given by

$$\langle u, v\rangle = \int_\Omega u(x)\, v(x)\, \mathrm{d}x + \int_\Omega \nabla u(x) \cdot \nabla v(x)\, \mathrm{d}x.$$

[1] Nonzero entries have numerical value 1.

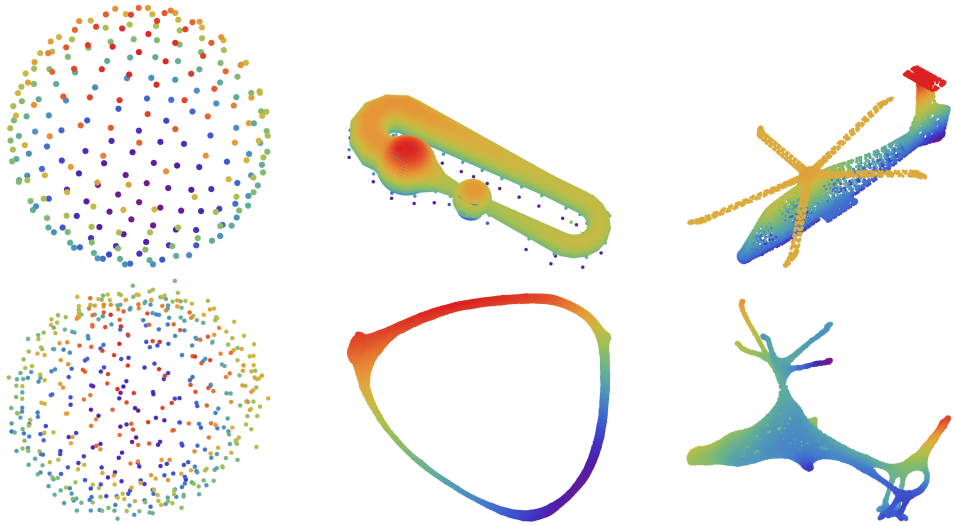

FIGURE 1.3. *From left to right: the matrices* `sphere3`, `pwt`, *and* `commanche_dual` *with their original meshes (top, see Example 1.4) and visual representations created using the techniques described in Section 2 (bottom, also compare with [24]).*

Let $g \in V$ be given. We consider the problem of finding an $f \in V$ such that

$$\Delta f(x) = -g(x), \qquad (x \in \Omega). \tag{1.1}$$

The first step in the finite-element method is to rewrite the above problem into its *weak formulation*.[2] Let $h \in V$ and suppose f is a solution to eqn (1.1), then using integration by parts

$$\int_\Omega g(x)\, h(x)\, \mathrm{d}x = -\int_\Omega \Delta f(x)\, h(x)\, \mathrm{d}x = -0 + \int_\Omega \nabla f(x) \cdot \nabla h(x)\, \mathrm{d}x.$$

Hence, every solution f necessarily satisfies the weak formulation of this problem:

$$a(f, h) = b(h), \qquad (h \in V), \tag{1.2}$$

where

$$a(u, v) := \int_\Omega \nabla u(x) \cdot \nabla v(x)\, \mathrm{d}x, \qquad b(u) := \int_\Omega g(x)\, u(x)\, \mathrm{d}x.$$

If we now choose functions $h_1, \ldots, h_m \in V$, we can look at the approximate solution ϕ to eqn (1.2) in the subspace $V_m := \langle h_1, \ldots, h_m \rangle_{\mathbf{R}} \subseteq V$ spanned by $h_1, \ldots, h_m$. Because $\phi \in V_m$, there exist coefficients $\phi_1, \ldots, \phi_m \in \mathbf{R}$ such that $\phi = \sum_{i=1}^m \phi_i\, h_i$, and therefore,

$$b(h_j) = a(\phi, h_j) = a\left(\sum_{i=1}^m \phi_i\, h_i, h_j\right) = \sum_{i=1}^m \phi_i\, a(h_i, h_j), \qquad (1 \le j \le m).$$

[2] If the weak formulation satisfies the conditions of the Lax–Milgram theorem, the solution f to the weak formulation is unique and hence also the solution to the original problem, provided such a solution exists [26].

So solving eqn (1.2) in V_m for ϕ amounts to solving the linear system

$$\sum_{i=1}^{m} a(h_i, h_j)\, \phi_i = b(h_j), \qquad (1 \le j \le m),$$

for $(\phi_1, \ldots, \phi_m) \in \mathbf{R}^m$. By choosing $h_1, \ldots, h_m$ such that their supports only have a small overlap we create a sparse matrix $A \in \mathbf{R}^{m \times m}$ with entries $a_{i\,j} := a(h_j, h_i)$ that are nonzero only for the $1 \le i, j \le m$ where $\operatorname{supp} h_i \cap \operatorname{supp} h_j \neq \emptyset$. This can be done, for instance, by triangulating Ω and choosing for each vertex i in the triangulation a function h_i with support contained in all triangles adjacent to the vertex i.

We can now directly create a visual representation for the various hypergraph representations of the matrix A (Table 1.1):

1. for the symmetric, bipartite, column-net, and row-net representations, we map each vertex i, r_i, and c_i to the center of $\operatorname{supp} h_i$ in $\mathbf{R}^d$,
2. for the finegrain representation, we map each vertex $v_{i\,j}$ to the center of $\operatorname{supp} h_i \cap \operatorname{supp} h_j$ in $\mathbf{R}^d$.

Thus, in this case, we can generate a visual representation from our original problem with little extra effort.

However, not all matrices have an immediate geometric origin (e.g. `twotone`), so we would like to be able to create such a visual representation ourselves if it cannot be provided directly. Hence, we will continue by discussing how to create such a visual representation in Section 2. In Section 3, we show how to use visual representations for hypergraph partitioning, and in Section 4 we apply this in the context of sparse LU decomposition. We conclude by implementing these methods and comparing them with both SuperLU and Mondriaan in Section 5.

2. Creating visual representations. To create visual representations, we employ the method described in [17, 24, 32], generalized to visualizing hypergraphs: we let the vertices of the hypergraph repel each other by an electrostatic-like force and let the hyperedges bind their vertices together like rubber bands. We model this as the minimization of an energy function where we lay out the graph in at least three dimensions (to prevent having to treat the special cases $d = 1$ and $d = 2$ where the form we propose for the energy function is not appropriate). For brevity, we will simply enumerate the vertices of the hypergraph we consider, thus assuming that $V = \{1, \ldots, k\}$ for some $k \in \mathbf{N}$.

DEFINITION 2.1. Let $G = (\{1, \ldots, k\}, \{e_1, \ldots, e_l\})$ be a finite weighted hypergraph with vertex weights $w_1, \ldots, w_k > 0$, hyperedge costs $c_1, \ldots, c_l > 0$, and let $d \ge 3$ be the dimension of the target space. Define

$$U := \{(x_1, \ldots, x_k) \in \mathbf{R}^{d \times k} \mid \forall 1 \le i < j \le k : x_i \neq x_j\}.$$

Then the *energy function of* G is defined as

$$f(x_1, \ldots, x_k) := \underbrace{\frac{\alpha}{2} \sum_{j=1}^{l} c_j \sum_{i \in e_j} \|x_i - z_j\|^\gamma}_{\text{rubber bands}} + \underbrace{\frac{\beta}{2} \sum_{j=1}^{k} \sum_{\substack{i=1 \\ i \neq j}}^{k} \frac{w_i\, w_j}{\|x_i - x_j\|^\delta}}_{\text{repelling charges}}, \tag{2.1}$$

for $(x_1, \ldots, x_k) \in U$, where $\alpha, \beta, \gamma, \delta > 0$ are constants, and for $1 \le j \le l$ the center of hyperedge e_j is defined as

$$z_j := \frac{1}{|e_j|} \sum_{i \in e_j} x_i.$$

Now, we will generate a visual representation by finding

$$\operatorname{argmin}\{f(x_1,\dots,x_k) \mid (x_1,\dots,x_k) \in U\}, \tag{2.2}$$

where f is the energy function of our hypergraph G. Approximate solutions to eqn (2.2) generated by our algorithm are shown in Fig. 1.3.

2.1. Constants. We can tinker with f by varying the constants α, β, γ, and δ. First, we can make a number of observations about U, f, and their symmetries.

1. $\{(R(x_1)+x,\dots,R(x_k)+x) \mid (x_1,\dots,x_k) \in U\} = U$ for all $x \in \mathbf{R}^d$ and all orthogonal transformations $R \in \mathrm{O}(\mathbf{R}^d)$.
2. U is an open subset of $\mathbf{R}^{d\times k}$ and $U = \theta\, U$ for all $\theta \in \mathbf{R}\setminus\{0\}$.
3. $f(R(x_1)+x,\dots,R(x_k)+x) = f(x_1,\dots,x_k)$ for all $x \in \mathbf{R}^d$, $R \in \mathrm{O}(\mathbf{R}^d)$.
4. $f \in C(U,[0,\infty[\,)$ is continuous; f is always bounded from below by 0; and if $k \geq 2$, f is unbounded from above.

So U and f are invariant under all translations and orthogonal transformations in $\mathbf{R}^d$, but while U is scaling-invariant, f in general is not.

Let $\theta > 0$ and $(x_1,\dots,x_k) \in U$. Then

$$f(\theta\, x_1,\dots,\theta\, x_k) = \frac{\alpha\,\theta^\gamma}{2}\sum_{j=1}^{l} c_j \sum_{i\in e_j} \|x_i - z_j\|^\gamma + \frac{\beta}{2\,\theta^\delta}\sum_{j=1}^{k}\sum_{\substack{i=1\\ i\neq j}}^{k} w_i\, w_j\, \|x_i - x_j\|^{-\delta}.$$

So in particular, for all $(x_1,\dots,x_k) \in U$ we have that

$$\frac{1}{\beta}\left(\frac{\beta}{\alpha}\right)^{\frac{\delta}{\gamma+\delta}} f\left(\left(\frac{\beta}{\alpha}\right)^{\frac{1}{\gamma+\delta}} x_1,\dots,\left(\frac{\beta}{\alpha}\right)^{\frac{1}{\gamma+\delta}} x_k\right) = \tilde{f}(x_1,\dots,x_k),$$

where $\tilde{f}$ is given by eqn (2.1), but with $\alpha = \beta = 1$. Therefore, we can pick $\alpha = \beta = 1$ without loss of generality; this just scales the minimum of f by an overall factor (a similar scaling property is derived in [24, Theorem 1]).

The function f is continuously differentiable for $\gamma \geq 2$. Calculating the partial derivatives of f, we find that for $1 \leq m \leq d$, $1 \leq n \leq k$,

$$\begin{aligned}\frac{\partial f(x_1,\dots,x_k)}{\partial x_{m\,n}} &= -\frac{\alpha\,\gamma}{2}\sum_{\{j \mid n\in e_j\}} \frac{c_j}{|e_j|}\sum_{i\in e_j} \|x_i - z_j\|^{\gamma-2}\,(x_{m\,i} - z_{m\,j})\\ &+ \frac{\alpha\,\gamma}{2}\sum_{\{j \mid n\in e_j\}} c_j\,\|x_n - z_j\|^{\gamma-2}\,(x_{m\,n} - z_{m\,j}) + \underbrace{\beta\,\delta\sum_{\substack{i=1\\ i\neq n}}^{k} \frac{w_i\, w_n}{\|x_i - x_n\|^{\delta+2}}\,(x_{m\,i} - x_{m\,n})}_{\text{repelling charges}}.\end{aligned} \tag{2.3}$$

Note that eqn (2.3) simplifies considerably if we pick $\gamma = 2$, as $\sum_{i\in e_j}\|x_i - z_j\|^{2-2}\,(x_{m\,i} - z_{m\,j}) = (\sum_{i\in e_j} x_{m\,i}) - |e_j|\,z_{m\,j} = 0$, which makes the first term disappear, and ensures that $f \in C^\infty(U,[0,\infty[)$ is smooth. This motivates us to pick $\gamma = 2$.

Calculating the repelling-charges part of eqn (2.3) for $n = 1,\dots,k$ requires $\mathcal{O}(k^2)$ evaluations which is too expensive for the hypergraphs we envision, with a huge number of vertices k. Luckily, we can circumvent this problem using techniques from

[24] and [29]: we will build the d-dimensional equivalent of an octree to group the repelling charges into clusters and treat far-away clusters of charges as a single, but heavier charge. To ensure that this works properly the location of this larger charge will be the weighted average location of the cluster and the weight of the charge will be set equal to the sum of the charge weights in the cluster. Note that the repelling-charges part of f in eqn (2.1) consists of applications of the map $x \mapsto \|x\|^{-\delta}$ for $x \in \mathbf{R}^d$, the Laplacian of which is given by $x \mapsto \delta\,(\delta - (d-2))\,\|x\|^{-\delta-2}$. Hence, we can ensure that this part of f is harmonic (i.e. with vanishing Laplacian) by choosing $\delta = d - 2$. This will in turn make our energy function behave well when treating clusters of far-away charges as a single, heavier charge, because of the *mean-value property* of harmonic functions (see [3, Theorem 1.6 and 1.24]).

Recapitulating the above:

1. we can pick $\alpha = \beta = 1$ since this only scales a solution to eqn (2.2),
2. we should pick $\gamma = 2$ to be able to calculate the rubber band contribution to eqn (2.3) efficiently,
3. we should pick $\delta = d - 2$ to be able to approximate the repelling charge contribution to eqn (2.3) by treating clusters of charges as a single, heavier charge.

Inserting these constants we obtain the following expressions for f and its partial derivatives:

$$f(x_1, \ldots, x_k) = \frac{1}{2} \sum_{j=1}^{l} c_j \sum_{i \in e_j} \|x_i - z_j\|^2 + \frac{1}{2} \sum_{j=1}^{k} \sum_{\substack{i=1 \\ i \neq j}}^{k} \frac{w_i\, w_j}{\|x_i - x_j\|^{d-2}}, \tag{2.4}$$

$$\frac{\partial f(x_1, \ldots, x_k)}{\partial x_{m\,n}} = \sum_{\{j \mid n \in e_j\}} c_j\,(x_{m\,n} - z_{m\,j}) + (d-2) \sum_{\substack{i=1 \\ i \neq n}}^{k} \frac{w_i\, w_n}{\|x_i - x_n\|^d}\,(x_{m\,i} - x_{m\,n}). \tag{2.5}$$

2.2. Connectedness. Before we start describing an algorithm to solve eqn (2.1), we should take care to ensure that such a solution actually exists.

THEOREM 2.2. Let $G = (V, E)$, U, and f be given as in Definition 2.1 and suppose G is non-empty.

Then precisely one of the following statements is true:

1. there exists a solution $(x_1, \ldots, x_k) \in U$ to eqn (2.2),
2. G is disconnected: we can write $V = V_1 \cup V_2$ as a disjoint union with $V_1, V_2 \neq \emptyset$, such that for all $e \in E$ we have $e \subseteq V_1$ or $e \subseteq V_2$.

Proof. Suppose G is disconnected. Without loss of generality, we can order the vertices such that $V_1 = \{1, \ldots, k'\}$ and $V_2 = \{k'+1, \ldots, k\}$. Suppose $(x_1, \ldots, x_k) \in U$ has minimum energy. Then for all $y \in \mathbf{R}^d \setminus \{0\}$ we have

$$\begin{aligned} &f(x_1 + y, \ldots, x_{k'} + y, x_{k'} - y, \ldots, x_k - y) \\ &\qquad = f(x_1, \ldots, x_k) + \beta \sum_{i=1}^{k'} \sum_{j=k'+1}^{k} \left(\frac{w_i\, w_j}{\|2\,y + x_i - x_j\|^{\delta}} - \frac{w_i\, w_j}{\|x_i - x_j\|^{\delta}} \right). \end{aligned}$$

This equation holds because all hyperedges $e_j \in E$ are either completely contained in V_1 or completely contained in V_2, such that all x_i with $i \in e_j$ are either translated by $+y$ or $-y$, respectively. This ensures that for all hyperedges $e_j \in E$ and vertices $i \in e_j$, the difference $x_i - z_j$ remains the same, which in turn leaves the first term of

eqn (2.1) unchanged. For the second term of eqn (2.1), we find that the difference $x_i - x_j$ only changes if $i \in V_1$ and $j \in V_2$, or vice versa.

For all $1 \leq i \leq k'$, $k' < j \leq k$, we have

$$\lim_{r\to\infty} \frac{w_i\, w_j}{\|2\,(r\,y) + x_i - x_j\|^\delta} = \lim_{r\to\infty} \frac{1}{|r|^\delta}\, \frac{w_i\, w_j}{\|2\,y + (x_i - x_j)/r\|^\delta} = 0$$

as $y \neq 0$ and $\delta > 0$. In particular there exists an $r > 0$ such that for all $1 \leq i \leq k'$, $k' < j \leq k$ we have

$$\frac{w_i\, w_j}{\|2\,(r\,y) + x_i - x_j\|^\delta} - \frac{w_i\, w_j}{\|x_i - x_j\|^\delta} < 0.$$

So for this r we have

$$f(x_1 + r\,y, \ldots, x_{k'} + r\,y, x_{k'+1} - r\,y, \ldots, x_k - r\,y) < f(x_1, \ldots, x_k),$$

hence $(x_1, \ldots, x_k)$ does not have minimum energy; we have reached a contradiction. Therefore, no solution to eqn (2.2) exists.

Suppose conversely that G has no disconnected components: for any disjoint union $V = V_1 \cup V_2$ with $V_1, V_2 \neq \emptyset$ there exists an $e \in E$ such that $e \cap V_1 \neq \emptyset$ and $e \cap V_2 \neq \emptyset$. So in particular for every $v, w \in V$ there exists a path of hyperedges $e_1, \ldots, e_n \in E$ such that $v \in e_1$, $w \in e_n$, and $e_{j-1} \cap e_j \neq \emptyset$ for all $1 < j \leq n$. We can see this by writing V as the disjoint union $\{v\} \cup (V \setminus \{v\})$ which gives us e_1 and continuing by induction to obtain e_{j+1} from the disjoint union $V = (e_1 \cup \ldots \cup e_j) \cup (V \setminus (e_1 \cup \ldots \cup e_j))$ until we reach w (which will happen eventually as each new hyperedge adds at least one new vertex and our hypergraph is finite).

Suppose that for a given $R > 0$, $(x_1, \ldots, x_k) \in U$ satisfies

$$\max_{1\leq i<j\leq k} \|x_i - x_j\| \geq R. \tag{2.6}$$

We will now focus on two vertices for which the relative distance in eqn (2.6) is maximal. As described above there exists a path of hyperedges between these two vertices. So there exist vertices $i_1, \ldots, i_{n+1} \in V$ and edges $e_1, \ldots, e_n \in E$ such that $\|x_{i_1} - x_{i_{n+1}}\|$ is maximal, $i_1 \in e_1$, $i_{n+1} \in e_n$, and $i_j \in e_{j-1} \cap e_j$ for $1 < j \leq n$. Along this path, we have

$$\begin{aligned} R \leq \|x_{i_1} - x_{i_{n+1}}\| &= \left\| x_{i_1} - z_1 + \sum_{m=2}^{n} (z_{j-1} - x_{i_m} + x_{i_m} - z_j) + z_n - x_{i_{n+1}} \right\| \\ &\leq \|x_{i_1} - z_1\| + \sum_{m=2}^{n} (\|x_{i_m} - z_{j-1}\| + \|x_{i_m} - z_j\|) + \|x_{i_{n+1}} - z_n\|. \end{aligned}$$

Hence, one of these $2\,n$ terms must be at least $R/(2\,n)$. So there exist a vertex $p \in V$ and a hyperedge $e_q \in E$ such that $p \in e_q$ and $\|x_p - z_q\| \geq R/(2\,n) \geq R/(2\,l)$. Therefore, f satisfies

$$f(x_1, \ldots, x_k) \geq \frac{\alpha}{2}\, c_q\, \|x_p - z_q\|^\gamma \geq \frac{\alpha}{2\,(2\,l)^\gamma} \left(\min_{1\leq j\leq l} c_j \right) R^\gamma. \tag{2.7}$$

Suppose that for a given $r > 0$, $(x_1, \ldots, x_k) \in U$ satisfies

$$\min_{1\leq i<j\leq k} \|x_i - x_j\| \leq r.$$

Fix $1 \leq i < j \leq k$ such that $\|x_i - x_j\|$ is minimal, then

$$f(x_1, \ldots, x_k) \geq \beta \frac{w_i\, w_j}{\|x_i - x_j\|^\delta} \geq \beta\, w_i\, w_j \frac{1}{r^\delta} \geq \beta \left(\min_{1 \leq i \leq k} w_i^2 \right) \frac{1}{r^\delta}. \tag{2.8}$$

Note that f is invariant under translations, so that without loss of generality we can restrict ourselves to solutions with $x_1 = 0$. Let

$$E := f((0, 0, \ldots, 0), (1, 0, \ldots, 0), (2, 0, \ldots, 0), \ldots, (k-1, 0, \ldots, 0))$$

be an upper bound for the minimum value of f.

By eqn (2.7) we know that there exists an $R > 0$ such that $f(0, x_2, \ldots, x_k) > E$ whenever $\max_{i \neq j} \|x_i - x_j\| > R$. On the other hand, by eqn (2.8) there exists an $r > 0$ such that $f(0, x_2, \ldots, x_k) > E$ whenever $\min_{i \neq j} \|x_i - x_j\| < r$. So if $(0, x_2, \ldots, x_k) \in U$ has minimal energy, then necessarily $(0, x_2, \ldots, x_k) \in C$ where

$$C := \bigcap_{1 \leq i < j \leq k} \left\{ (x_1 = 0, x_2, \ldots, x_k) \in \mathbf{R}^{d \times k} \mid r \leq \|x_i - x_j\| \leq R \right\}.$$

As $C \subseteq \mathbf{R}^{d \times k}$ is both closed and bounded (because $x_1 = 0$), C is compact by Theorem 1.8.17 in [16]. We also have that $C \subseteq U$ and f is continuous on U, so f is continuous on the compact set C and therefore $f|_C$ attains its minimum at a certain point in C by Theorem 1.8.8 of [16]. Since the minimum of f necessarily lays within C by construction, this is a solution to eqn (2.2). □

Therefore, in solving eqn (2.2), we should restrict ourselves to connected hypergraphs. If the provided hypergraph is not connected, we have to treat each connected component separately, where a solution is guaranteed to exist by Theorem 2.2.

Algorithm 1 Determines the connected components of a hypergraph $G = (V, E)$, with $V = \{1, \ldots, k\}$. Vertices v and w belong to the same connected component of G iff $p_v = p_w$. This algorithm is adapted from [10, Section 21.3].

```
1:  for all v ∈ V do
2:      p_v ← v;
3:  for all e ∈ E do
4:      for all v ∈ e do
5:          while p_v ≠ p_{p_v} do
6:              p_v ← p_{p_v};
7:      p ← min{p_v | v ∈ e};
8:      for all v ∈ e do
9:          p_{p_v} ← p;
10: for all v ∈ V do
11:     while p_v ≠ p_{p_v} do
12:         p_v ← p_{p_v};
```

2.3. Algorithm. We use the above observations to create a multilevel algorithm which approximates a minimum-energy solution. Firstly, we find the connected components via Algorithm 1. This algorithm works by creating a forest, i.e. a collection of rooted trees, where the parent of a vertex $v \in V$ is denoted by $p_v \in V$, and each root satisfies $p_v = v$. The algorithm merges the trees of all vertices contained in a

Algorithm 2 Finds solutions to eqn (2.2) for a given connected hypergraph $G = (V, E)$, adopted from [24].

1: $G^0 \leftarrow G$; $j \leftarrow 0$;
2: **while** G^j is too large **do**
3: coarsen G^j to G^{j+1} with surjective $\pi^j : V_{G^j} \to V_{G^{j+1}}$;
4: $j \leftarrow j + 1$;
5: let x^j be a random visual representation for G^j
6: (so pick $x^j(v) \in \mathbf{R}^d$ at random for all vertices $v \in V_{G^j}$);
7: **while** $j \geq 0$ **do**
8: improve x^j by Algorithm 3;
9: **if** $j > 0$ **then**
10: **for all** vertices $v \in V_{G^{j-1}}$ **parallel do**
11: $x^{j-1}(v) \leftarrow$ scale $x^j(\pi^j(v)) +$ small random displacement;
12: $j \leftarrow j - 1$;

hyperedge, for all hyperedges. After this is completed, each vertex is directly attached to its root, which represents its connected component.

Secondly, we apply Algorithm 2 to each connected component to generate a visual representation. We create a hierarchy of coarsenings of the hypergraph and visual representations for these coarser hypergraphs, to avoid getting stuck in local energy minima. Here, we follow the graph visualization algorithm from [24]. We coarsen hypergraphs by creating a maximal matching that is constructed greedily from heavy (high cost) hyperedges, as heavy hyperedges have the tendency to pull the vertices contained in them closer together when we try to find the energy minimum, and then merging the matched vertices. To prevent star hypergraphs from forming (which disrupt this matching procedure [12]) we also always match single-neighbor vertices to their neighbor. Coarsening a hypergraph G to a hypergraph H in this way, we obtain a surjective map $\pi : V_G \to V_H$ which maps each collection of matched vertices of G to a single vertex of H. We also merge hyperedges $e, e' \in E_G$ satisfying $\pi(e) = \pi(e')$ (where $\pi(e) = \{\pi(v) \mid v \in e\}$) to a single hyperedge and set the cost of this new hyperedge to the sum of the costs of e and e'. At line 10 of Algorithm 2, we introduce the **parallel do** construct. We use it to denote parallel for-loops that can directly be parallelized because their iterations are independent.

For the coarsest version of the hypergraph, we start out with a random visual representation, which is then improved by Algorithm 3 using the steepest descent method. After the visual representation has been improved for the coarse hypergraph we scale it and add small random displacements, to ensure that different vertices in the fine hypergraph do not occupy the same position. Thus, we obtain a visual representation for the fine hypergraph. This layout is then again improved by Algorithm 3. We continue doing this until we obtain a visual representation for the original hypergraph.

Algorithm 3 gives the details of the improvement procedure on line 8 of Algorithm 2. The part of the gradient of the energy function f belonging to x_n, is denoted by y_n. We create a tree T which recursively groups the points belonging to the visual layout; this tree consists of nodes $t \in T$ which have position $x_t \in \mathbf{R}^d$ (the average position of all points contained in t) and weight $w_t > 0$ (the sum of the weights of all vertices contained in t). The functionality of siblings of nodes in the tree is extended by letting the root have a dummy node as sibling (denoting the end of the

Algorithm 3 Improves a given approximate solution $(x_1, \ldots, x_k) \in U$ to eqn (2.2) for a hypergraph $G = (\{1, \ldots, k\}, \{e_1, \ldots, e_l\})$ using steepest descent.

while we are not satisfied with the solution **do**
  build tree T recursively clustering $x_1, \ldots, x_k$;
  for $n = 1$ **to** k **parallel do**
    $y_n \leftarrow 0$;
  for $j = 1$ **to** l **parallel do**
    $z_j \leftarrow \frac{1}{|e_j|} \sum_{i \in e_j} x_i$;
  for $n = 1$ **to** k **parallel do**
    for all j such that $n \in e_j$ **do**
      $y_n \leftarrow y_n + c_j \, (x_n - z_j)$; (cf. eqn (2.5))
  for $n = 1$ **to** k **parallel do**
    $t \leftarrow$ root of T;
    while $t \neq$ dummy **do**
      if x_n is far away from t **then**
        $y_n \leftarrow y_n + (d-2) \, w_t \, w_n \, \|x_t - x_n\|^{-d} \, (x_t - x_n)$; (cf. eqn (2.5))
        $t \leftarrow$ sibling of t;
      else if t has a child **then**
        $t \leftarrow$ child of t;
      else
        for all $i \in t$, $i \neq n$ **do**
          $y_n \leftarrow y_n + (d-2) \, w_i \, w_n \, \|x_i - x_n\|^{-d} \, (x_i - x_n)$; (cf. eqn (2.5))
        $t \leftarrow$ sibling of t;
  determine appropriate stepsize $\alpha > 0$;
  for $n = 1$ **to** k **parallel do**
    $x_n \leftarrow x_n - \alpha \, y_n$;

tree traversal), and letting nodes without siblings have the sibling of their parent as sibling. This facilitates a fast, direct tree traversal without backtracking [29]. During steepest descent, we determine the step size by comparing the bounding box volume to the individual gradients of the vertices to prevent blowup for the first few steps and then decrease the found stepsize by multiplying it by 0.9 [24]. Note that apart from the building of the tree grouping the hypergraph vertices, almost all parts of Algorithm 3 consist of d-dimensional floating point arithmetic that is directly parallellizable over all vertices and hyperedges. This makes Algorithm 3 suitable for many-core architectures such as GPUs.

3. Partitioning. Suppose that $x_1, \ldots, x_k \in \mathbf{R}^d$ is a visual representation of our hypergraph $G = (\{1, \ldots, k\}, \{e_1, \ldots, e_l\})$, obtained either directly or by using the methods from Section 2. Then we will use this spatial layout to create a partitioning of the hypergraph in a desired number of $m \in \mathbf{N}$ parts. To do so, we employ the `k-means++` method [2], given by Algorithm 4. This algorithm searches for m centers $z_1, \ldots, z_m \in \mathbf{R}^d$ such that

$$\sum_{i=1}^{k} \min_{1 \leq j \leq m} \|x_i - z_j\|^2 \tag{3.1}$$

is minimal, which is NP-hard for all $m \geq 2$ [1]. The advantage of `k-means++` is that the algorithm finds $z_1, \ldots, z_m$ in $\mathcal{O}(k \, m \, \log m)$ time, while the value of eqn (3.1) is

expected to be within a factor of $8\,(\log m + 2)$ from its minimum value already at the start (line 6) of the first iteration [2]. The `k-means++` algorithm therefore permits us to isolate clusters quickly in the visual representation of our hypergraph, which correspond to highly interconnected patches of vertices in the hypergraph. Algorithm 4 is furthermore easily parallelized in shared memory (lines 3 and 7). The sum at line 11 can also be performed in parallel by computing partial sums for the z_j and summing these afterwards.

Algorithm 4 The `k-means++` algorithm [2] finds centers $z_1, \ldots, z_m \in \mathbf{R}^d$ for a given set of points $x_1, \ldots, x_k \in \mathbf{R}^d$, trying to minimize eqn (3.1).

```
1:  set z_1 to a randomly chosen point from {x_1, ..., x_k};
2:  for n = 2 to m do
3:     for i = 1 to k parallel do
4:        d_i ← min_{1≤j<n} ‖x_i − z_j‖^2;
5:     choose z_n to be equal to x_i with probability d_i/(d_1 + ... + d_k);
6:  for a fixed number of iterations do
7:     for i = 1 to k parallel do
8:        j_i ← argmin_{1≤j≤m} ‖x_i − z_j‖^2;
9:     for j = 1 to m do
10:       C_j := {1 ≤ i ≤ k | j_i = j};
11:       z_j := (1/|C_j|) Σ_{i∈C_j} x_i;
```

The m disjoint subsets $C_1, \ldots, C_m \subseteq V$ produced by Algorithm 4 form an m-way partitioning of V. It should be remarked that this way of generating a partitioning does not enforce balancing of the partitioning, but in general `k-means++` does a good job of dividing the point set into parts of approximately equal size: large groups of points pull centers harder towards themselves, which enlarges the other, smaller, groups. Such balancing can be observed in Fig. 1.1 and Fig. 5.1. Partitionings generated by Algorithm 4 can further be improved by subjecting them to a few iterations of the Kernighan–Lin algorithm [27].

4. LU decomposition. We will now apply the ideas discussed in the previous sections to performing an LU decomposition of a given matrix in parallel using nested dissection [18, 22], see also [4, 9, 20, 25, 28].

DEFINITION 4.1. Let $A \in \mathbf{C}^{m\times m}$ be a given $m \times m$ matrix. Then a *(permuted) LU decomposition* of A is a decomposition of the form

$$P\,A\,Q = L\,U, \tag{4.1}$$

where $P, Q \in \{0,1\}^{m\times m}$ are permutation matrices and $L, U \in \mathbf{C}^{m\times m}$ with entries $l_{i\,j}$, $u_{i\,j}$ respectively, such that $l_{i\,j} = 0$ for $i < j$, $u_{i\,j} = 0$ for $i > j$, and $l_{i\,i} = 1$ for all i.

We include permutations in Definition 4.1 to ensure that well-behaved matrices like $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ also have an LU decomposition. Calculating an LU decomposition for a given matrix with complete pivoting is described by Algorithm 5. While this algorithm is fine for small dense matrices, we get into trouble with large sparse matrices for two reasons: Algorithm 5 takes $\mathcal{O}(m^3)$ iterations (which scales rather badly), regardless of matrix sparsity, and as illustrated by Example 4.2, the sparsity of the original matrix may be lost during the decomposition, requiring up to $\mathcal{O}(m^2)$ memory. We will remedy these problems by calculating appropriate permutation matrices P and

Algorithm 5 LU decomposition (algorithm 3.4.2 from [19]). Determines for a matrix $A \in \mathbf{C}^{m\times m}$ with entries $a_{i\,j}$ factors P, Q, L, and U such that $P\,A\,Q = L\,U$.

```
 1: for k = 1 to m do
 2:    π_k ← k, σ_k ← k;
 3: for k = 1 to m do
 4:    find k ≤ i, j ≤ m such that |a_{i j}| is maximal;
 5:    swap rows k and i and swap π_k and π_i;
 6:    swap columns k and j and swap σ_k and σ_j;
 7:    if a_{k k} ≠ 0 then
 8:       for i = k + 1 to m do
 9:          a_{i k} ← a_{i k}/a_{k k};
10:       for i = k + 1 to m do
11:          for j = k + 1 to m do
12:             a_{i j} ← a_{i j} − a_{i k} a_{k j};
13: set p_{i j} to 1 if π_i = j and 0 otherwise, to form P;
14: set q_{i j} to 1 if σ_j = i and 0 otherwise, to form Q;
15: set l_{i j} to a_{i j} if i > j, 1 if i = j, and 0 otherwise, to form L;
16: set u_{i j} to a_{i j} if i ≤ j and 0 otherwise, to form U;
```

Q, using the techniques from the previous sections.

EXAMPLE 4.2. Consider the following decomposition (using Algorithm 5 without pivoting):

$$\begin{pmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 0 & 0 \\ 1 & 0 & 2 & 0 \\ 1 & 0 & 0 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{1}{2} & 1 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{3} & 1 & 0 \\ \frac{1}{2} & -\frac{1}{3} & -\frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 & 1 \\ 0 & \frac{3}{2} & -\frac{1}{2} & -\frac{1}{2} \\ 0 & 0 & \frac{4}{3} & -\frac{2}{3} \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Here, the sparsity pattern of the original matrix is lost completely in the L and U factors, and a *fill-in* of six new nonzeros is created. Suppose we permute the matrix by swapping the first and last rows and columns, then we obtain the following decomposition:

$$\begin{pmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 & 1 \\ 0 & 2 & 0 & 1 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & \frac{1}{2} \end{pmatrix}.$$

Now, L and U have the same sparsity pattern as the original matrix. Furthermore, performing Algorithm 5 on the permuted matrix also required less work because of zeros that were preserved during decomposition: the two loops around line 12 can skip most rows and columns because either $a_{i\,k}$ or $a_{k\,j}$ is equal to 0.

4.1. Recursive BBD form. Following Example 4.2 we will try to bring the sparse matrix into Bordered Block Diagonal (BBD) form [25] as illustrated in Fig. 4.1 (a). The block in the lower-right corner is commonly called the *Schur complement*. A row that contains a nonzero in a column that intersects the first diagonal block and also in a column that intersects the second diagonal block is said to be *cut* or *split* with respect to the subdivision of the matrix. Cut columns are defined similarly. Performing Algorithm 5 on a matrix in such a form will only generate fill-in in the

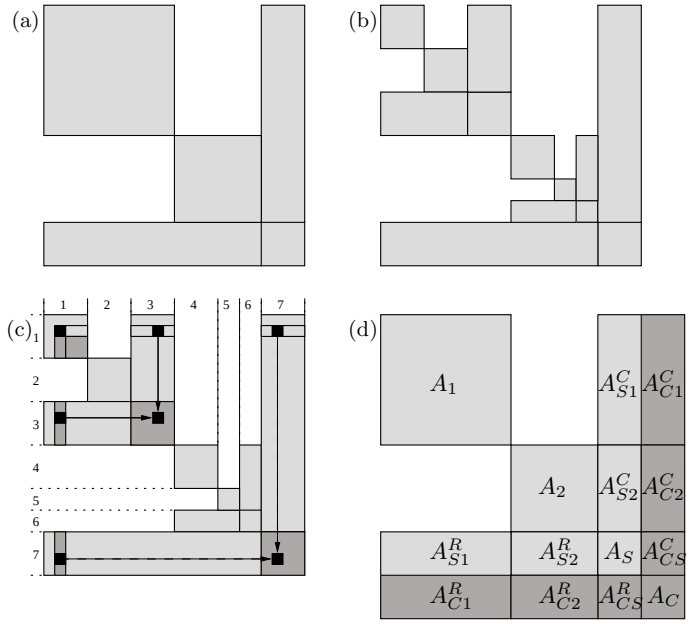

FIGURE 4.1. *Nested dissection for LU decomposition. (a) Bordered Block Diagonal (BBD) matrix form; (b) recursive BBD matrix form; (c) LU decomposition contributions; (d) matrix form for Algorithm 6.*

shaded blocks and the costly inner loop can skip the empty blocks. By doing this recursively, see Fig. 4.1 (b), the amount of fill-in will be further reduced. This principle is called *nested dissection* [18].

Such a recursive matrix layout furthermore permits us to create a parallel LU decomposition algorithm (similar to [28]). We illustrate this in Fig. 4.1 (c) where we are busy performing Algorithm 5 along the diagonal of diagonal block 1 (for the sake of simplicity we do not perform any pivoting). For this nonzero on the diagonal, performing LU decomposition will only modify the darkly shaded parts of the matrix and therefore leave the diagonal blocks 2, 4, and 5 untouched. Furthermore, the LU decomposition contributions of all diagonal blocks to the Schur complements 3, 6, and 7 do not depend on the actual values in the Schur complements, so we can perform LU decomposition on blocks 1, 2, 4, and 5 in parallel and add the contributions to the Schur complements afterwards. To process the nonzero on the diagonal of block 1, we need nonzero values not only from the cut rows and columns of Schur complement 3 (solid arrows), but *also* from the rows and columns of Schur complement 7 (dashed arrows). So we need to keep track of *all* previous Schur complements during parallel LU decomposition.

For the parallel LU algorithm outlined in Algorithm 6, we therefore work recursively on a matrix of the form illustrated in Fig. 4.1 (d). Here, we added *contribution blocks* A_C, A^C_{C1}, A^C_{C2}, A^C_{CS}, A^R_{C1}, A^R_{C2}, and A^R_{CS} to the matrix, which are indicated by a darker shade. At the start of Algorithm 6 (so for the original matrix A), these are empty, but as the algorithm further recurses, the contribution blocks will contain all the contributions of the LU decomposition to Schur complements of the previous recursion levels (i.e. the data required for the dashed arrows in Fig. 4.1 (c)). In an implementation of Algorithm 6 it would be efficient to store the nonzeros $a_{i\,j}$ of the

matrix by increasing $\min\{i, j\}$. Thus, we keep all data necessary to perform the LU decomposition at line 6 together, regardless of the level of recursion. This is better than storing the nonzeros by increasing i (Compressed Row Storage, CRS) or j.

Algorithm 6 Parallel recursive LU decomposition of the matrix from Fig. 4.1 (d).

1: **if** we wish to continue subdividing **then**
2: apply Algorithm 6 recursively and in parallel to

A_1	A^C_{S1}	A^C_{C1}
A^R_{S1}	0	0
A^R_{C1}	0	0

and

A_2	A^C_{S2}	A^C_{C2}
A^R_{S2}	0	0
A^R_{C2}	0	0

3: add the contributions from A_1 and A_2 to A_S, A_C, A^R_{CS}, and A^C_{CS};
4: perform LU decomposition (e.g. Algorithm 5) on

A_S	A^C_{CS}
A^R_{CS}	A_C

,

only permuting within A_S and stopping after factorizing A_S;
5: **else**
6: perform LU decomposition on A, only permuting within the lightly shaded part of the matrix and stopping after factorizing A_1, A_2, and A_S;

It is important to note that the LU decompositions performed in Algorithm 6 on line 4 and line 6 are incomplete in the sense that they only treat part of the given matrix. In Algorithm 5, this would amount to specifying a number $1 \leq m' < m$ and letting k run from 1 to m' at line 3 and choosing i and j such that $k \leq i, j \leq m'$ at line 4. To improve performance, multifrontal [11] or supernodal [13] methods could be used to perform these LU decompositions. The condition at line 1 in Algorithm 6 is used to stop the algorithm whenever the resulting diagonal block becomes so small that a direct LU decomposition would outperform further recursion, or when there is a risk of the diagonal matrices becoming singular. Note that Algorithm 6 is *processor-oblivious* in the sense that we can continue recursing on the diagonal blocks while there are still more processors available, up to the recursion depth where the diagonal blocks are still sufficiently large.

Since with this parallel method we can only pivot within each block (not doing so would destroy the recursive BBD form), we could encounter a singular submatrix, as illustrated by Theorem 4.3 and Example 4.4.

THEOREM 4.3. Let $A \in \mathbf{C}^{a\times a}$, $B \in \mathbf{C}^{b\times b}$, $C \in \mathbf{C}^{c\times c}$ such that $d := a-(b+c) \geq 0$ and A is of the form

$$A = \begin{pmatrix} \begin{array}{|c|c|} \hline B & 0 \\ \hline 0 & C \\ \hline \end{array} & \begin{array}{|c|} \hline \\ \hline \\ \hline \end{array} \\ \begin{array}{|cc|} \hline & \\ \hline \end{array} & \begin{array}{|c|} \hline D \\ \hline \end{array} \end{pmatrix}.$$

If $\det(A) \neq 0$, then

$$b + c - d \leq \operatorname{rank}(B) + \operatorname{rank}(C) \leq b + c. \tag{4.2}$$

Proof. First of all, note that if the matrix $A' \in \mathbf{C}^{a \times a}$ is obtained from A using Gauss–Jordan elimination with column pivoting, then $\det(A) = 0$ if and only if $\det(A') = 0$. Suppose that $\det(A) \neq 0$, then by performing these operations on B and C separately, we find the nonzero value

$$\det \begin{pmatrix} \boxed{I_{\operatorname{rank}(B)}} & \square & 0 & 0 & \square \\ 0 & 0 & 0 & 0 & \\ 0 & 0 & \boxed{I_{\operatorname{rank}(C)}} & \square & \\ 0 & 0 & 0 & 0 & \\ 0 & \square & 0 & \square & \boxed{D} \end{pmatrix} = \pm \det \begin{pmatrix} \boxed{I_{\operatorname{rank}(B)}} & 0 & 0 & \square & \square \\ 0 & \boxed{I_{\operatorname{rank}(C)}} & \square & 0 & \\ 0 & 0 & 0 & 0 & \\ 0 & 0 & 0 & 0 & \\ 0 & 0 & \square & \square & \boxed{D} \end{pmatrix}$$

$$= \pm \det \begin{pmatrix} 0 & 0 & \square \\ 0 & 0 & \\ \square & \square & \boxed{D} \end{pmatrix}.$$

The resulting smaller matrix has size $a - \operatorname{rank}(B) - \operatorname{rank}(C)$ and must be of maximum rank because its determinant is nonzero. Let $e := a - \operatorname{rank}(B) - \operatorname{rank}(C) - d$, then $e \geq a - b - c - d = 0$. If $e \leq d$, a matrix with the above nonzero pattern can have maximum rank $e + d$. If $e > d$, the rank of such a matrix can be at most $2\,d < e + d$. Therefore, it is necessary that $e \leq d \iff a - \operatorname{rank}(B) - \operatorname{rank}(C) - d \leq d \iff a - 2\,d \leq \operatorname{rank}(B) + \operatorname{rank}(C) \iff (b + c + d) - 2\,d \leq \operatorname{rank}(B) + \operatorname{rank}(C)$, from which eqn (4.2) follows. □

Theorem 4.3 shows us that we cannot assume our diagonal blocks to be invertible whenever the Schur complement is nonempty. Furthermore, it motivates us to reduce the size of the Schur complement: this will increase the minimum rank that the diagonal blocks are required to have and thereby increases stability. In terms of hypergraph partitioning we therefore see that we should at all times try to make the Schur complement *as small as possible*: this will increase parallelism in the sense that more rows/columns can be treated in parallel by Algorithm 6, it will reduce fill-in, and it will improve stability.[3]

To prevent the diagonal blocks from becoming singular we allow for an optional specification of a desired (strengthened) matrix diagonal beforehand [15], which will be preserved by the generated permutations as described in Section 4.2. As Example 4.4 shows however, this is not guaranteed to solve the problem.

Example 4.4. Let

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 & 1 \\ 4 & 2 & 0 & 0 & 1 \\ 0 & 0 & 2 & 1 & 1 \\ 0 & 0 & 1 & 2 & 1 \\ 1 & 1 & 1 & 1 & 2 \end{pmatrix}, \quad B = \begin{pmatrix} 2 & 1 \\ 4 & 2 \end{pmatrix}, \quad C = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}.$$

Then $a = 5$, $b = 2$, $c = 2$, $d = 1$, $\operatorname{rank}(B) = 1$, $\operatorname{rank}(C) = 2$, and $\det(A) = 3 \neq 0$. Therefore, the bound in eqn (4.2) is tight: $b + c - d = 3 = \operatorname{rank}(B) + \operatorname{rank}(C)$. Note that $\det(B) = 0$ even though the product of the diagonal elements of A is maximal.

[3] So hypergraph partitioners used for the purpose of bringing the matrix into recursive BBD form should use the *cut-net* metric instead of the $(\lambda - 1)$-metric, reducing the number of cut hyperedges, and not the associated communication volume.

During the performed benchmark with SuperLU (Table 5.1) we found that for 8 of the 28 matrices no pivoting was required at all (not even in the Schur complements), and in all other cases pivoting with a threshold of 10^{-6} was sufficient. Therefore, if we keep the Schur complements small, strengthen the matrix diagonal, and use threshold pivoting within diagonal blocks and Schur complements, we anticipate that submatrix singularity will not pose any significant problems in practice.

4.2. Permutations. We will now apply the ideas from sections 1.1, 2, and 3 to obtain the desired permutations to bring a given sparse matrix $A \in \mathbf{C}^{m \times m}$ with nz nonzeros into recursive BBD form. This method will be referred to as *visual matrix ordering* (VMO). We assume the matrix to be square, because we want to use VMO for LU decomposition.

Firstly, we need to determine what kind of hypergraph we will use to represent A (from Table 1.1). Using only the symmetric representation is not appropriate, because LU factorization is usually applied to unsymmetric matrices. The column-net and row-net approach often do not yield optimal partitionings when compared to the finegrain representation [8]. However, the finegrain representation results in nz vertices, thus degrading the performance of Algorithm 3, which would scale as $\mathcal{O}(nz \log(nz) + m)$. Inspection of the visual layouts revealed that a good layout for the finegrain representation could be obtained by laying out its dual (the bipartite representation, which is a graph) and then mapping each nonzero to the average of the points of the row and column belonging to that particular nonzero. As the bipartite representation has only $2\,m$ vertices instead of nz, the layout can be generated much faster, scaling as $\mathcal{O}(m \log(m) + nz)$. It also permits us easily to maintain a previously selected strengthened diagonal in the generated permutations

Therefore, let $G = (V, E)$ be the bipartite representation of our sparse matrix A. To avoid the problem of ending up with a singular matrix during recursion, we permit a desired strengthened diagonal to be specified with the matrix, in the form of a perfect bipartite graph matching $M \subseteq E$, [15]. We will view this matching as a map $\mu : V \to V$ which maps each vertex $v \in V$ to $\mu(v) \in V$ such that the edge $\{v, \mu(v)\} \in M$ (as M is a perfect matching, exactly one vertex $\mu(v)$ has this property).

We can also incorporate the values of the nonzeros of the matrix in the partitioning by setting the edge costs of G to $|a_{i\,j}|$ for each edge $\{i, j\} \in E$ (optionally rescaling these values to a fixed interval to avoid convergence issues in Algorithm 3). This is natural, because zeros of the matrix are not incorporated at all in eqn (2.1) (as they are not included in E), so letting the edge cost of $\{i, j\}$ go to 0 as $|a_{i\,j}| \to 0$ gradually decreases the influence of $\{i, j\}$ on the energy function of G to zero. However, as this reduced the quality of the partitionings in terms of fill-in, we did not use this option for the performed experiments.

Using Algorithm 2 we generate a visual representation of G. Then we apply Algorithm 7 to obtain $V = V_1 \cup V_2 \cup V_3$ and $E = E_1 \cup E_2 \cup E_3$, where $p(v)$ and $q(e)$ denote the part indices for vertices v and edges e. Algorithm 7 turns the edge separator, obtained by partitioning the vertices using Algorithm 4, into a vertex separator. To do so we exploit the geometry of the partitioning by letting the vertices closest to the plane (described in Algorithm 7 by its normal r and distance δ on line 7) separating the two groups of vertices be chosen to be added to the vertex separator. We ensure that we preserve the strengthened diagonal in lines 11 and 13: this ensures that edges from the matching M are contained completely in either V_1, V_2, or V_3, which prevents them from entering the darker off-diagonal blocks in Fig. 4.2. This can be skipped if no matching is available or desired (e.g. in the context of

Algorithm 7 Given a graph $G = (V, E)$ with visual representation $x : V \to \mathbf{R}^d$ and a map $\mu : V \to V$ derived from a perfect bipartite graph matching $M \subseteq E$, this algorithm partitions V and E into V_1, V_2, V_3 and E_1, E_2, E_3 respectively, where no $\{v, w\} \in E$ exists with $v \in V_1$ and $w \in V_2$, and such that $e \in E_1 \to e \subseteq V_1$, $e \in E_2 \to e \subseteq V_2$, and $e \in E_3 \to e \cap V_3 \neq \emptyset$ (Fig. 4.2 (left)).

1: determine two centers $z_1, z_2 \in \mathbf{R}^d$ in $x(V) \subseteq \mathbf{R}^d$ by Algorithm 4;
2: **for all** $v \in V$ in parallel **do**
3: **if** $\|x(v) - z_1\| \leq \|x(v) - z_2\|$ **then**
4: $p(v) \leftarrow 1$;
5: **else**
6: $p(v) \leftarrow 2$;
7: $r \leftarrow z_2 - z_1$; $\delta \leftarrow \frac{1}{2}(z_2 + z_1) \cdot r$;
8: **for all** $e = \{v, w\} \in E$ **do**
9: **if** $\{p(v), p(w)\} = \{1, 2\}$ **then**
10: **if** $|x(v) \cdot r - \delta| \leq |x(w) \cdot r - \delta|$ **then**
11: $p(v) \leftarrow 3$; $p(\mu(v)) \leftarrow 3$;
12: **else**
13: $p(w) \leftarrow 3$; $p(\mu(w)) \leftarrow 3$;
14: **for all** $e = \{v, w\} \in E$ in parallel **do**
15: $q(e) \leftarrow \max\{p(v), p(w)\}$;
16: sort the vertex pairs $\{v, \mu(v)\}$ by their p-values to obtain V_1, V_2, V_3;
17: sort the edges by their q-values to obtain E_1, E_2, E_3;

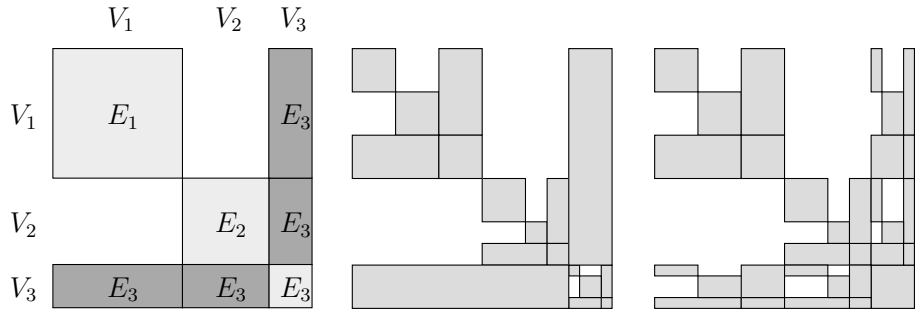

FIGURE 4.2. *Matrix partitioning for Algorithm 7 (left) and improved partitionings obtained by bringing either the Schur complement (middle) or the cut rows and columns (right) into BBD form.*

sparse matrix–vector multiplication instead of LU decomposition), which will result in smaller V_3 and E_3. However, for LU decomposition it is necessary, in particular to maintain square blocks on the diagonal. After the first iteration of Algorithm 7, we again apply it to $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ and continue doing this recursively to obtain recursive BBD permutations for our matrix A as shown in the rightmost column of Fig. 1.1.

The permutations themselves can directly be obtained from the recursive partitioning of V: the rows and columns of the block E_1 (see Fig. 4.2) are exactly the vertices in V_1, and similarly for the rows and columns of the blocks E_2 and E_3. Therefore, a simple linear walk through the reordered vertices (line 16 of Algorithm 7) will provide the proper permutations of the rows and columns of our matrix A.

When permuting the matrix to recursive BBD form, we have additional freedom

in permuting the rows and columns of V_3 in Fig. 4.2 (left) (also see Fig. 5.1). A direct way to do this is also to apply Algorithm 7 recursively to $G_3 = (V_3, E_3')$, just like we do for G_1 and G_2. Here E_3' consists of all edges $e \in E_3$ satisfying $e \subseteq V_3$ (so E_3' is the lightly shaded E_3 part of Fig. 4.2 (left)). This gives permutations as illustrated in Fig. 4.2 (middle). An advantage of this method is that the strengthened diagonal is also maintained within the Schur complement.

Another way in which the additional freedom can be used, is to bring the cut rows and columns into recursive BBD form as illustrated in Fig. 4.2 (right). Doing this is a little more tricky: first of all, we assign a two-bit number to each vertex in V_3, initially set to 00. We also keep track of the edges $E_{1\,3}$ between V_1 and V_3, and edges $E_{2\,3}$ between V_2 and V_3. Now, if V_i $(i = 1, 2)$ is split with Algorithm 7 into $V_{i\,1}$, $V_{i\,2}$, and $V_{i\,3}$ we can loop through all edges $\{v, w\}$ in $E_{i\,3}$ with $w \in V_3$. Then if $v \in V_{i\,1}$ we set the first bit of the number associated with w and if $v \in V_{i\,2}$ we set the second bit. If we do this for both splits of V_1 and V_2, and then sort the vertices in V_3 by their two-bit numbers we obtain a permutation as shown in Fig. 4.2 (right). By expanding these numbers to b two-bit pairs and keeping track of the edges extending to the Schur complements for up to b splits, we can bring the cut rows and columns into recursive BBD form up to the bth level.

5. Experiments. We implemented the VMO algorithm in C++ using the Intel Threading Building Blocks library for many-core parallelism where we chose to generate visual representations in $d = 4$ dimensions to be able to perform all parallel vector calculations in Algorithm 3 and Algorithm 4 efficiently on either the CPU (one Streaming SIMD `xmm*` register for a point in $\mathbf{R}^4$) or the GPU (a `float4` register for a point in $\mathbf{R}^4$). This furthermore ensures that we do not need to take a square root in eqn (2.5).

To measure the quality of the generated permutations we compared VMO to the SuperLU [13] LU decomposition package by measuring fill-in, see Table 5.1. In this case we made use of the additional freedom in the cut rows and columns by also recursively subdividing the cut parts of the graph while retaining the strengthened diagonal (Fig. 4.2 (middle) and Fig. 5.1 (left)) to ensure that few small pivots are encountered along the diagonal. We performed four decompositions for each matrix where we used permutations generated by VMO, as well as the built-in COLAMD(A), MMD($A^T + A$), and MMD($A^T A$) column permutations generated by SuperLU. For the permutations generated by SuperLU we retained the default SuperLU 4.1 options, while for the VMO permutations we first performed a run without any pivoting and then a run with threshold pivoting[4], using a value of $u = 10^{-6}$. To ensure we would not run into numerical problems we used a strengthened diagonal obtained via a heavy edge matching in the bipartite representation of A, augmented to a perfect matching via the Hopcroft–Karp algorithm [23]. We furthermore validated the decomposition by comparing calculated condition numbers for all permutation methods and letting SuperLU calculate the backward error of the solution to $A\,x = b$ obtained by solving the system using the decomposition $A = L\,U$ (section 3.1 of [19]) for $b = A\,(1, \ldots, 1)^T$. Table 5.1 shows that in 8 of the 28 cases, decomposition of the matrices permuted by VMO did not require *any* pivoting at all, not even in the Schur complements. From the table we see that VMO compares favorably with SuperLU: looking at the lowest fill-in of COLAMD(A), MMD($A^T + A$), and MMD($A^T A$) and the fill-in of VMO for

[4]SuperLU performs row pivoting by generating a row permutation π such that for all $1 \le j \le m$, $|a_{\pi(j)\,j}| \ge u \max_{1 \le i \le m} |a_{i\,j}|$ where $u \in [0, 1]$ is the desired threshold, see [14, eqn (4.4.7)].

Matrix	Size	Nonzeros	VMO	CMD	MMD+	MMD×
swang1	3169	20841	**6.2**	7.7	6.7	8.2
lns_3937	3937	25407	**15.0**	17.5	132.2	17.5
poli_large	15575	33074	**1.6**	1.6	1.6	1.6
mark3jac020*	9129	56175	68.1	45.6	121.3	**43.9**
fd18*	16428	63406	**21.9**	24.1	302.0	25.5
lhr04*	4101	82682	6.0	**4.1**	20.6	4.3
raefsky6	3402	137845	**2.7**	3.4	4.5	3.1
shermanACb*	18510	145149	19.0	45.3	**14.3**	57.2
bayer04*	20545	159082	10.2	4.2	41.8	**4.2**
Zhao2*	33861	166453	158.1	115.1	1280.1	**107.0**
mult_dcop_03	25187	193216	3.1	**2.0**	3.4	5.9
jan99jac120sc*	41374	260202	71.8	**15.9**	52.4	19.7
bayer01*	57735	277774	7.5	**5.4**	47.6	5.6
sinc12*	7500	294986	37.8	44.7	**36.3**	45.3
onetone1*	36057	341088	32.1	14.4	149.0	**14.2**
mark3jac140sc*	64089	399735	**111.0**	125.7	4435.0	152.0
af23560	23560	484256	**24.8**	25.0	82.7	26.9
e40r0100*	17281	553562	9.2	9.2	137.5	**8.4**
sinc15*	11532	568526	56.3	58.0	**48.7**	57.2
Zd_Jac2_db*	22835	676439	9.6	**5.1**	32.1	5.7
lhr34c*	35152	764014	7.0	4.7	50.5	**4.7**
sinc18*	16428	973826	**65.7**	67.8	68.2	72.3
torso2	115967	1033473	10.2	16.8	**8.2**	14.5
twotone	120750	1224224	35.8	**15.2**	1448.1	17.0
lhr71c*	70304	1528092	6.7	4.8	66.4	**4.7**
av41092*	41092	1683902	64.6	26.0	177.6	**23.8**
bbmat*	38744	1771722	32.0	**26.7**	1000.6	26.8

TABLE 5.1

Comparison between VMO and SuperLU 4.1 in terms of fill-in, defined as $(nz(L) + nz(U) - nz(I))/nz(A)$ *for* $A = LU$*. The best result for each matrix is* ***bold****, CMD = COLAMD, MMD+ =* $MMD(A^T + A)$*, and MMD×* $= MMD(A^T A)$ *are the column pre-orderings determined by SuperLU. Matrices marked with * required threshold* 10^{-6} *pivoting for VMO.*

each of the 28 test matrices, we find that on average the fill-in of VMO equals 1.52 times the lowest fill-in of the other methods, and that VMO outperforms all other methods in 8 cases. This indicates that the permutations generated by VMO are useful in the context of sparse LU decomposition.

We also compared VMO with Mondriaan [31] in terms of matrix partitioning. Firstly, we did this in the context of cache-oblivious sparse matrix–vector multiplication where the matrices are permuted into recursive Separated Block Diagonal (SBD) form [33] (with the cut rows and columns in the middle instead of at the end) to decrease the number of cache-misses, independent of the particular cache hierarchy of the processor performing the multiplication. Results are further improved by also using the additional freedom in the cut rows and columns to bring these into recursive SBD form (Fig. 5.1 (right)). We measure the matrix multiplication time with the same program and on the same processor as [34]: a single node of the Huygens supercomputer equipped with a dual-core 4.7GHz IBM Power6+ processor with 64kB L1 cache per core, a semi-shared L2 cache of 4MB, and an L3 cache of 32MB on

which the matrix–vector multiplication program has been compiled with the IBM XL compiler. In Table 5.2, we compare the matrix–vector multiplication time for the original matrix with the best result from [34] (where the matrix has been permuted by Mondriaan), and with the result obtained by using VMO.

Matrix	Rows	Columns	Nonzeros	Orig.	[34]	VMO
ex37	3565	3565	67591	0.116	0.113	0.113
memplus	17758	17758	126150	0.308	0.300	0.280
rhpentium	25187	25187	258265	0.645	0.515	0.646
lhr34	35152	35152	764014	1.37	1.34	1.34
lp_nug30	52260	379350	1567800	5.35	4.85	9.15
s3dkt3m2	90449	90449	1921955	7.81	7.27	7.80
tbdlinux	112757	21067	2157675	6.43	2.36	5.66
stanford	281903	281903	2312497	19.0	9.35	5.88
stanford_berkeley	683446	683446	7583376	20.9	19.2	22.5
wikipedia-20051105	1634989	1634989	19753078	249	116	128
cage14	1505785	1505785	27130349	69.4	74.4	99.0
wikipedia-20060925	2983494	2983494	37269096	688	256	264

TABLE 5.2

Comparison with Mondriaan in the context of cache-oblivious sparse matrix–vector multiplication [34]. We compare the original matrix–vector multiplication time with the best time from [34] (which used Mondriaan 3.01 for reordering) and the best time with VMO.

VMO performs poorly for lp_nug30 and cage14. For lp_nug30, this can be explained by a lack of underlying geometrical structure: the visual representation of this matrix is a featureless blob from which little extra information can be obtained, resulting in quite bad permutations. The matrix cage14 already possesses a nonzero layout that is well suited for matrix–vector multiplication: both Mondriaan and VMO fail to improve the matrix–vector multiplication time. For tbdlinux, wikipedia-20051105, and wikipedia-20060925 VMO shows improvements comparable to those of Mondriaan, while for memplus and stanford the results are even better. As generating the permutations with VMO is much faster (see Table 5.3), this makes VMO a viable alternative to Mondriaan in this context.

Another comparison with Mondriaan was made in terms of the cut-net metric, which is the appropriate metric in the context of LU decomposition because of Theorem 4.3. Hence, we look at the maximum number of cut rows and columns in all matrix (sub)divisions. While Mondriaan divides the matrix among a given number of processors, VMO continues subdividing the matrix until it can no longer continue. Therefore, we ran Mondriaan with a hybrid splitting strategy for the cut-net metric to divide the matrix into 256 parts with a permitted imbalance of 0.1 to obtain permutations comparable to those of VMO.

In Table 5.3, we measure the time it takes Mondriaan to perform the matrix partitioning and divide this by the time it takes VMO to generate a visual representation of the matrix and to generate a partitioning from this visual representation. All timings except for those marked with * were measured on a system with a quad-core 2.8GHz Intel Core i7 860 processor and 8GB RAM, in particular to illustrate the gains of using VMO on a many-core system. The entries marked with * needed to be benchmarked on a different system, because of Mondriaan's memory requirements: these were performed on a dual quad-core 2.4GHz AMD Opteron 2378 system with 32GB RAM. From Table 5.3 we find that VMO is on average 21.6 times faster than

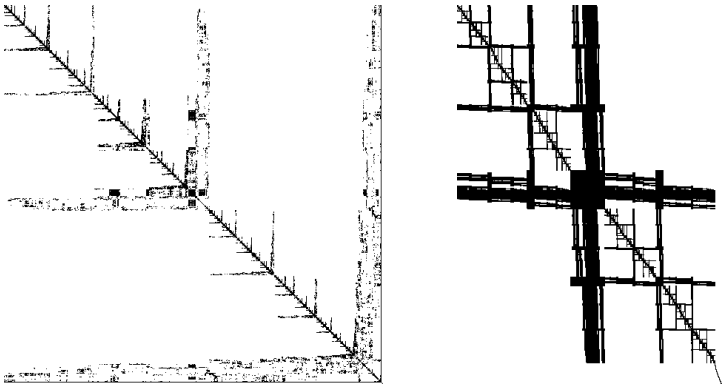

FIGURE 5.1. *The matrices* rhpentium *(left) and* wikipedia-20070206 *(right), permuted by VMO to recursive BBD and recursive SBD form, respectively. Additional permutation freedom is used for* rhpentium *as in Fig. 4.2 (middle), and for* wikipedia-20070206 *as in Fig. 4.2 (right).*

Matrix	Speedup V + O	Speedup O	Relative cut
ex37	13.4	53.2	0.39
memplus	2.1	11.0	2.86
rhpentium	14.5	57.0	1.08
lhr34	6.5	39.8	2.47
lp_nug30	9.1	144.7	2.20
s3dkt3m2	4.7	26.7	1.02
tbdlinux	25.0	228.3	0.75
stanford	7.9	78.6	2.37
stanford_berkeley	17.0	118.0	3.26
wikipedia-20051105	36.0	290.7	0.61
cage14*	4.3	29.1	0.62
wikipedia-20060925*	119.4	1104.3	1.02

TABLE 5.3

*Comparison with Mondriaan in terms of the calculation time and the largest number of cut rows/columns in a split. Speedup is defined as the time required by Mondriaan 3.01 to perform the matrix partitioning, divided by the time required by VMO to generate both the visual representation and permutations (V + O) or just the permutations (O). The last column gives the maximum of the number of cut rows and columns in all splits of VMO, divided by the maximum obtained by Mondriaan. Entries marked with a * were benchmarked on a different system because of a lack of memory. The test matrices are the same as those from Table 5.2.*

Mondriaan, and if for all matrices a visual representation would already have been given the average speedup would even be 181.8. We also measure the maximum of the number of cut rows and columns in all subdivisions of the matrix for VMO and divide this by the maximum for Mondriaan. This gives a measure for the relative maximum cut size when comparing the two methods: the maximum cut size obtained by VMO is on average 1.55 times that of Mondriaan and in four cases it is less. To make the comparison as fair as possible we used Mondriaan with the cut-net metric for partitioning, but it should still be remarked that minimizing the maximum number of cut rows and columns is not the primary objective of Mondriaan and the balancing restrictions placed on Mondriaan are absent for VMO.

6. Conclusion. We have shown that it is possible to create and use the visual representations of hypergraphs to generate partitionings and orderings which are of sufficient quality for sparse LU decomposition (Table 5.1) and sparse matrix–vector multiplication (Table 5.2). Our method generates orderings on average 21.6 times faster than Mondriaan (Table 5.3). We generalized the 2D/3D graph visualization method from [24] to generate hypergraph geometries in higher dimensions. Furthermore, the algorithms to generate visual representations (Algorithm 3) and matrix orderings (Algorithm 7) are well suited to shared-memory many-core parallel architectures such as current many-core CPUs and GPUs. We have implemented these algorithms in the software package VMO.

This also opens up opportunities for further research, such as moving from a shared-memory parallel architecture to distributed-memory, which would require significant modifications of Algorithm 3, Algorithm 7, and the data structures involved. Since VMO is fast and parallel, it also has potential to remove computational partitioning bottlenecks in large applications such as the human bone simulations in [5].

Acknowledgments. We thank Albert-Jan Yzelman for performing the sparse matrix–vector multiplication experiments (Table 5.2). We thank Job Kuit, Joop Kolk, and Paul Zegeling for helpful discussions and comments. We thank the Dutch supercomputing center SARA in Amsterdam and the Netherlands National Computing Facilities foundation NCF for providing access to the Huygens supercomputer.

REFERENCES

[1] D. Aloise, A. Deshpande, P. Hansen, and P. Popat, *NP-hardness of Euclidean sum-of-squares clustering*, Machine Learning, 75 (2009), pp. 245–248.

[2] D. Arthur and S. Vassilvitskii, *k-means++: the advantages of careful seeding*, in SODA '07: Proceedings of the 18th annual ACM-SIAM symposium on Discrete algorithms, Philadelphia, PA, 2007, SIAM, pp. 1027–1035.

[3] S. Axler, P. Bourdon, and W. Ramey, *Harmonic function theory*, vol. 137 of Graduate Texts in Mathematics, Springer-Verlag, New York, 1992.

[4] C. Aykanat, A. Pinar, and U. V. Çatalyürek, *Permuting sparse rectangular matrices into block-diagonal form*, SIAM J. Sci. Comput., 25 (2004), pp. 1860–1879.

[5] C. Bekas, A. Curioni, P. Arbenz, C. Flaig, G. H. van Lenthe, R. Müller, and A. J. Wirth, *Extreme scalability challenges in micro-finite element simulations of human bone*, Concurrency Computat.: Pract. Exper., 22 (2010), pp. 2282–2296.

[6] C. Berge, *Graphs and hypergraphs*, North-Holland, Amsterdam, revised ed., 1976.

[7] U. V. Çatalyürek and C. Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Trans. Par. Dist. Syst., 10 (1999), pp. 673–693.

[8] ———, *A fine-grain hypergraph model for 2D decomposition of sparse matrices*, in Proceedings 8th International Workshop on Solving Irregularly Structured Problems in Parallel, IEEE Press, Los Alamitos, CA, 2001, p. 118.

[9] U. V. Çatalyürek, C. Aykanat, and E. Kayaaslan, *Hypergraph partitioning-based fill-reducing ordering*, Technical Report OSUBMI-TR-2009-n02, Department of Biomedical Informatics, Ohio State University, Columbus, OH, April 2009.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*, MIT Press, Cambridge, MA, third ed., 2009.

[11] T. A. Davis and I. S. Duff, *An unsymmetric-pattern multifrontal method for sparse LU factorization*, SIAM J. Matrix Anal. Appl., 18 (1997), pp. 140–158.

[12] T. A. Davis and Y. F. Hu, *The University of Florida sparse matrix collection.* ACM Trans. Math. Software (to appear), 2010.

[13] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu, *A supernodal approach to sparse partial pivoting*, SIAM J. Matrix Anal. Appl., 20 (1999), pp. 720–755.

[14] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods for Sparse Matrices*, Monographs on Numerical Analysis, Oxford University Press, Oxford, UK, 1986.

[15] I. S. Duff and J. Koster, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM J. Matrix Anal. Appl., 22 (2001), pp. 973–996.
[16] J. J. Duistermaat and J. A. C. Kolk, *Multidimensional Real Analysis I: Differentiation*, Cambridge University Press, Cambridge, UK, 2004.
[17] T. M. J. Fruchterman and E. M. Reingold, *Graph drawing by force-directed placement*, Softw. Pract. Exper., 21 (1991), pp. 1129–1164.
[18] A. George, *Nested dissection of a regular finite element mesh*, SIAM J. Numer. Anal., 10 (1973), pp. 345–363.
[19] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, 3rd ed., 1996.
[20] L. Grigori, E. G. Boman, S. Donfack, and T. A. Davis, *Hypergraph-based unsymmetric nested dissection ordering for sparse LU factorization*, SIAM J. Sci. Comput., 32 (2010), pp. 3426–3446.
[21] B. Hendrickson and T. G. Kolda, *Graph partitioning models for parallel computing*, Parallel Comput., 26 (2000), pp. 1519–1534.
[22] B. Hendrickson and E. Rothberg, *Improving the run time and quality of nested dissection ordering*, SIAM J. Sci. Comput., 20 (1998), pp. 468–489.
[23] J. E. Hopcroft and R. M. Karp, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225–231.
[24] Y. F. Hu, *Efficient and high quality force-directed graph drawing*, The Mathematica Journal, 10 (2005), pp. 37–71.
[25] Y. F. Hu, K. C. F. Maguire, and R. J. Blake, *A multilevel unsymmetric matrix ordering algorithm for parallel process simulation*, Comput. Chem. Engrg., 23 (2000), pp. 1631–1647.
[26] C Johnson, *Numerical solution of partial differential equations by the finite-element method*, Cambridge University Press, Cambridge, UK, 1987.
[27] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal, 49 (1970), pp. 291–307.
[28] M. R. Mehrabi and R. A. Brown, *An incomplete nested dissection algorithm for parallel direct solution of finite element discretizations of partial differential equations*, J. Sci. Comput., 8 (1993), pp. 373–387.
[29] N. Nakasato, *Oct-tree method on GPU: $42/Gflops cosmological simulation.* arXiv:0909.0541v1 [astro-ph.IM], 2009.
[30] S. Parter, *The use of linear graphs in Gauss elimination*, SIAM Rev., 3 (1961), pp. 119–130.
[31] B. Vastenhouw and R. H. Bisseling, *A two-dimensional data distribution method for parallel sparse matrix-vector multiplication*, SIAM Rev., 47 (2005), pp. 67–95.
[32] C. Walshaw, *A multilevel algorithm for force-directed graph-drawing*, J. Graph Algorithms Appl., 7 (2003), pp. 253–285.
[33] A. N. Yzelman and R. H. Bisseling, *Cache-oblivious sparse matrix–vector multiplication by using sparse matrix partitioning methods*, SIAM J. Sci. Comput., 31 (2009), pp. 3128–3154.
[34] ———, *Two-dimensional cache-oblivious sparse matrix–vector multiplication.* Preprint, 2010.