A PARALLEL MULTISTAGE ILU FACTORIZATION BASED ON A HIERARCHICAL GRAPH DECOMPOSITION *

PASCAL HÉNON † AND YOUSEF SAAD ‡

Abstract. PHIDAL (Parallel Hierarchical Interface Decomposition ALgorithm) is a parallel incomplete factorization method which exploits a hierarchical interface decomposition of the adjacency graph of the coefficient matrix. The idea of the decomposition is similar to that of the well-known wirebasket techniques used in domain decomposition. However, the method is devised for general, irregularly structured, sparse linear systems. This paper describes a few algorithms for obtaining good quality hierarchical graph decompositions and discusses the parallel implementation of the factorization procedure. Numerical experiments are reported to illustrate the scalability of the algorithm and its effectiveness as a general purpose parallel linear system solver.

Key words. Parallel Incomplete LU factorization, ILU, Sparse Gaussian Elimination, Wirebasket decomposition, Interface decomposition, Preconditioning, ILU with threshold, ILUT, Iterative methods, Sparse linear systems.

AMS subject classifications. 65F10, 65N06.

1. Introduction. A number of Incomplete LU factorization techniques have been developed in recent years which attempt to combine some of the general-purpose features of standard ILU preconditioners with the good scalability attributes of multi-level methods. Examples include the ILU with Multi-elimination (ILUM, [14]), the MLILU [3], and MRILU algorithms [4], and the Algebraic Recursive Multilevel Solver (ARMS) [17] and its parallel version pARMS [9]. An ingredient that is common to all these methods is to reorder the linear system to enable a parallel (incomplete) elimination of groups of unknowns. Solution algorithms which implement similar strategies within domain decomposition have also been developed and can be found, for example, in the pARMS [9] package. This paper presents a new technique in this category. The main attraction of the new method is that it exploits a static 'hierarchical' decomposition of the graph which yields natural parallelism in the factorization process. This is in contrast with the methodology used in pARMS in which the independent sets and levels are defined dynamically. The underlying ordering on which the method is based is reminiscent of the 'wirebasket' techniques of domain decomposition methods [18, 19], in which cross points in a domain partitioning play an important role. Cross points have played a special role in many domain decomposition techniques, see [5] for example. One can also view the methods from the angle of an ILU factorization combined with a form of nested dissection ordering in which cross points in the separators play a special role. In this regard, the method bears some resemblance with the FETI method [6]. The ideas discussed in this paper also bear resemblance with those sketched by Magolu Monga Made and van der Vorst [12, 10, 11], for matrices arising from finite difference discretizations of 2-D partial differential equations.

The algorithm is based on defining a 'hierarchical interface structure'. The hierarchical decomposition starts with a partitioning of the graph, with one layer of overlap. Then, 'stages' or 'levels' are defined from this partitioning, with each level consisting of a set of groups or connectors (small connected subgraphs). The levels obey an important property which is that each connector of a given level is a separator for connectors of a lower level. Thus, for a simple 5-point graph the connectors at level one are the sets of interior nodes for each subdomain obtained from the partitioning. Level two includes the interfaces between domains at the exclusion of cross points, and level three consists of all the cross points. The vertices are globally ordered by level, in increasing order of level number. The incomplete factorization process proceeds by level from lowest to highest. Because of the separation property of the connectors at different levels, it turns out that this process can be made highly parallel by tuning the dropping strategy to the levels. More precisely, the dropping strategies will attempt to maintain the independent set structure extracted from the interface decomposition.

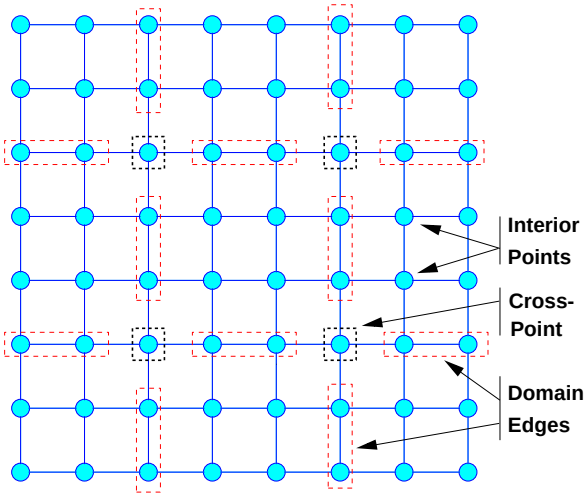
As was already mentioned, a major difference between this method and existing methods of this type, such as pARMS, is that the preordering used for parallelism, is performed at the outset instead of dynamically. This has both advantages and disadvantages. Because the ordering is static, the factorization is rather fast.

*This work was supported in part by the NSF under grants NSF/ACI-0305120 and NSF/INT-0003274, and by the Minnesota Supercomputer Institute.

†INRIA Futurs, Scalapplix project - LaBRI, 351 cours de la Libération 33405 Talence, France, henon@labri.fr

‡University of Minnesota, 200 Union Street S.E., Minneapolis, MN 55455. email: saad@cs.umn.edu

FIG. 2.1. *Partition of a* 8×8 *5-point mesh into 9 subdomains and the corresponding hierarchical interface structure.*



On the other hand, the ordering may face difficulties in the case when, for example, zero or small pivots are encountered. In pARMS, some form of diagonal pivoting is performed to bypass this difficulty.

The main ingredient of the ILU factorization is the hierarchical interface decomposition of the graph. It is fairly easy to find definitions which will achieve the desired goals of yielding a hierarchy of subgraphs which separate each other. For example, just assigning to each node a label which is defined as the number of subdomains to which the node belongs, yields, under certain conditions a decomposition into levels of independent sets, given by the labels. This is the case for simple graphs such as the one of a 5-point finite difference matrix. Thus, in the illustration of Figure 2.1, this strategy would yield 3 different labels. Nodes labeled "1" are interior nodes, nodes labeled "2" will be interface nodes (excluding cross points) and nodes labeled "4" are the cross points. Cross points are separators for interfaces which are separators for the sets of interior nodes for each domain. This is just the desired decomposition.

This simple strategy does not always work. It will fail for example if a 9-point discretization is used. In such situations, there are usually ways to modify the labels to yield the desired hierarchical decomposition. However, decompositions obtained in this way are often suboptimal for "rich" graphs, in that they may yield a large number of levels. This is an important issue to which a substantial part of the paper is devoted.

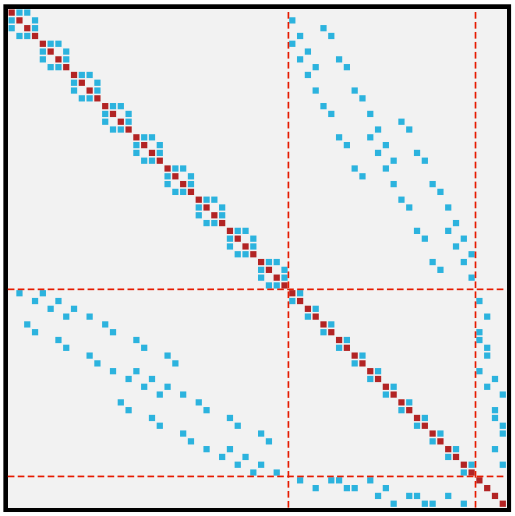
The next section discusses the hierarchical decomposition of a graph. Section 3 presents the details of the parallel elimination process. Section 6 reports on some experiments. The paper ends with some concluding remarks and tentative ideas for future work.

2. Hierarchical Interface Decomposition. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be the adjacency graph of a sparse matrix A which has a symmetric pattern so that $\mathcal{G}$ is undirected. Assume that $\mathcal{G}$ is partitioned into p subgraphs. An edge-based partitioning is used, in the sense that edges are primarily assigned to subdomains, and therefore vertices at the interfaces overlap two or more domains. This is the common situation which arises, for example, when elements in finite elements methods, are assigned to processors.

We begin by illustrating the hierarchical decomposition process on a matrix A which arises from a simple 5-point finite difference discretization of a Laplacean in 2-dimensional space. The vertex set $\mathcal{V}$ consists of three types of vertices representing the common "interior vertices", "domain-edges", and "cross-points" of the partition, see Figure 2.1 for an illustration. These three types of points are grouped in sets which will be referred to as "connectors". Thus, the interior points of a given domain form one connector, and the domain-edges, i.e., the set of interface points linking two given subdomains (excluding cross-points) also form a connector. In this case, levels of connectors are also naturally defined. Level one connectors are the sets of interior points, level 2 connectors are the domain-edges, and the level 3 connectors are the cross points.

This decomposition of the vertices has the interesting property that different connectors at the same level are not coupled. Another interesting property is that the higher level connectors separate the lower

FIG. 2.2. *Matrix associated with a* 8×8 *5-point mesh reordered according to the hierarchical interface decomposition.*



level connectors: interior domains (level 1 connectors) are separated by domain-edges and cross-points (level 2 and level 3 connectors) and domain-edge are themselves separated by cross-points. These properties can be exploited in a parallel context by taking advantage of the block structure of the matrix A obtained by reordering the unknowns according to this decomposition. If the unknowns are reordered according to their level numbers, from the lowest to highest, the block structure of the reordered matrix would be as shown in Figure 2.2. In this figure we can see that for a 5-point mesh, this kind of ordering will naturally yield a diagonal block structure for each level of connectors. Notice that the block structure leads to natural parallelism if an ILU decomposition is performed provided fill-in is avoided where it would hamper parallel processing of the factorization. Our goal is to generalize this ordering to irreguraly structured graphs in order to obtain a similar matrix structure which can be exploited in a parallel solver. It is worth mentioning that the regularity of the block structure in Figure 2.2 (all connector of a level have the same size) is related to the regularity of the mesh and the particular partitioning used. Similarly, the tridiagonal structure of the diagonal blocks is also due to the nature of the problem. It is not the goal of the ordering to achieve such a regular structure. Once the hierarchical decomposition is found, the ordering of the unknowns will label in sequence the unknowns in each connector. Connectors of level l are labeled before those of level $l+1$, for any level l.

The remainder of this section will discuss extensions of the "wirebasket" structure which was just seen for the 5-point mesh example to irregularly structured graphs. We will refer to this structure as a "hierarchical interface decomposition" of the graph. Given the general definition, we will then propose in the next section, some heuristics to compute the interface decomposition.

We begin with a formal definition of a hierarchical decomposition of a graph. We will call a *connector* a connected subgraph of $\mathcal{G}$. A *connector decomposition*, or *interface decomposition*, is a grouping of connectors into *levels*. So, a *level* is simply a set of connectors. Levels are labeled from 1 to p. Two connectors are *adjacent* (or "coupled") if there is at least one edge linking a vertex from one connector to a vertex of the other. A *hierarchical interface decomposition* is defined as follows.

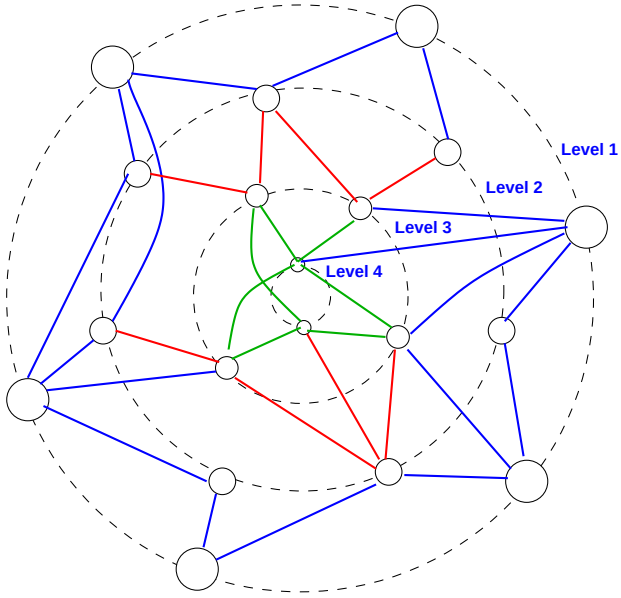
DEFINITION 2.1. *A set of levels labeled* $1, \ldots, p$ *is a hierarchical decomposition of* $\mathcal{G}$*, if it satisfies the following conditions:*

1. *The connectors are disjoint and altogether they form a partition of the graph,*
2. *Connectors of the same level are not adjacent,*
3. *Connectors of a given level* $l > 1$ *are separators for connectors at level* $l-1$*. This means that the*

removal of a connector of level l will disconnect (in the original graph) at least two connectors of level $l-1$.

Condition (2) by itself implies that two connectors C_1, C_2 which are not adjacent in the original graph and which belong to the same level l, must be separated by some connector C_3 from a different level. Condition (3) simply states that the level number of C_3 must be $l+1$. An illustration of the definition is given in Figure 2.3.

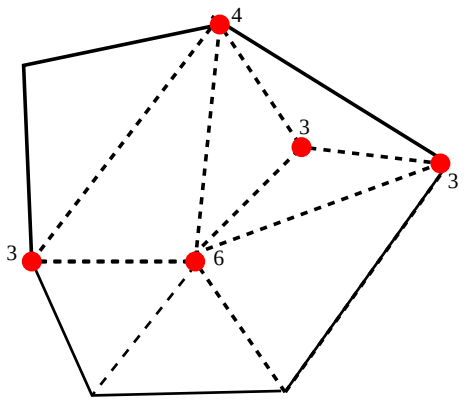
FIG. 2.3. *An example of a hierarchical decomposition of a graph. The circles are connectors and the links show the adjacencies between them. The connectors are grouped into 4 levels indicated by the dashed concentric circles.*



The issue now is to obtain such hierarchical decompositions. We may attempt to develop algorithms specifically for this task. In other words, we could seek algorithms that would take a graph and produce a partition into connectors satisfying the above conditions. In addition, the algorithm would also attempt to yield a good load balancing. This is not attempted in this paper. Instead, we will use as a starting point a partitioned graph, where the partitioning results from a standard graph partitioner such as METIS [8] or Scotch [13].

A few additional clarifications regarding the above definition may be helpful. The term 'interface' is used only because of the nature of the algorithms that will produce the decomposition. The term 'hierarchical' refers to the levelization of the connector sets with the property that the connectors of a given level l separate those connectors of lower levels $m < l$. For the 5-point example a good levelization was achieved by defining the level number of a connector from the number of adjacent domains. For general irregular domain and partitions, the situation is more complex and this process, when it works, will yield a poor levelization in the sense that the resulting levelization may produce levels which contains few connectors and thus produce a high number of levels. Figure 2.4 shows an irregular domain partition (the figure only shows the connectors, not the graph vertices and edges). In this case, there are five levels. Clearly, a better level-structure of the connectors is achieved by grouping all the connectors represented as dots together, resulting in only 3 sets of independent connectors. In subsection 2.3, we discuss an algorithm able to extract a good level partitioning.

FIG. 2.4. *Connector levels resulting from an irregular domain partition. The solid lines represent the domain boundaries. The dashed lines represent level-2 connectors. The big dots are the level-i connectors with $i > 2$. The level labels for these connectors are shown next to them.*



Before continuing, it may be insightful to consider alternatives for obtaining hierarchical interface decompositions. A first viewpoint can be realized by looking at the matrix in Figure 2.2 which is very similar to one that would be obtained from multicoloring. In fact it can be viewed as a "block" multicoloring of the graph, in the same way that the orderings considered in ARMS [17] can be viewed as block versions of independent set orderings. While this is correct, the main issue is to define connectors first from the partitioned graph. Once these are obtained then a multicoloring process can be called to define the levels. The point is that the difficulty resides in getting good connectors, not in getting the levels.

2.1. Algorithms for building hierarchical decompositions of a graph. We begin with some definitions and notation that will be used in the remainder of the paper. We are given an initial partitioning of the graph with overlap. The algorithm to define the hierarchical decomposition for general graphs will exploit the freedom to reassign the interface nodes to the neighboring subdomains so that certain properties of the modified partitioning are satisfied. In order to do this in a systematic manner, we associate with each vertex u a unique key $Key(u)$

$$Key(u) = \{i_1, .., i_m\}$$

which contains a list of all subdomains i_k to which u belongs in the graph partitioning. It is to be understood here that the assignment of u to subdomains will be altered dynamically by the procedure which defines the decomposition and so the key will be changing in the course of the procedure. We also denote by $C_{i_1,\ldots,i_m}$ the set of vertices which are overlapped by subdomains $i_1, i_2, \cdots, i_m$. In particular note that

$$Key(u) = \{i_1, .., i_m\} \Leftrightarrow u \in C_{i_1,\ldots,i_m}.$$

The subsets $C_{i_1,\ldots,i_m}$ are termed connectors. The notation $Key(C) = \{i_1, .., i_m\}$ will also be used for the connector $C = C_{i_1,..,i_m}$. We will also denote by $deg(C)$ the cardinality of the key of C, i.e.,

$$deg(C) = |Key(C)|$$

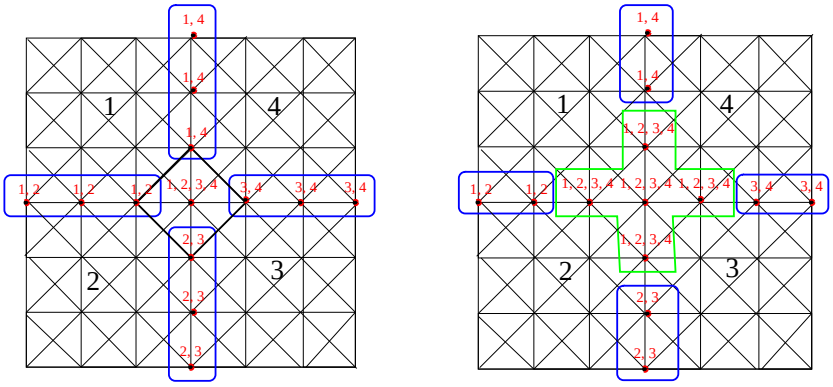
which is the number of subdomains that include C. Note that C is an intersection of subdomains, so there is no ambiguity in this definition. Similarly, we can also define the *domain degree* of a node by $deg(u) = |Key(u)|$, which is again the number of domains with which u is associated. By abuse of language we will sometimes call this the degree of u, which is to be distinguished from the actual degree of u in the graph.

The purpose of the keys is to help build good connectors. Given an arbitrary set of domain labels $\{i_1, i_2, .., i_k\}$, one can formally define any connector $C_{i_1,i_2,..,i_k}$ as the set of all nodes belonging to subdomains $i_1, \ldots, i_k$ (most connectors defined this way will be empty). For any particular partitioning, $Key(u) = \{i_1, \ldots, i_k\}$ iff u belongs to $C_{i_1,i_2,..,i_k}$. The connectors depend on the particular partitioning. The algorithms to be described in this section will slightly alter the partitioning, i.e., the keys, or equivalently, the connectors,

to achieve certain properties. In the proposed algorithms, these slight modifications will affect only the overlapping regions of the subdomains.

In the example of the grid, the partition of the interface into domain-edges and cross-points defines a partition of connectors: interior domain are connectors of degree 1, domain-edges are connectors of degree 2, and cross-points are connectors of degree 4. If C is a domain-edge between domain 1 and 2 then $Key(C) = \{1, 2\}$. As was mentioned above, in more complex cases, it will be necessary to redefine keys in order for the connectors to be independent. Though Section 2.2 is devoted to this issue we provide an example in Figure 2.5 to illustrate the idea.

FIG. 2.5. *(a) Partition of a 9-points grid into 4 subdomains. Connectors induced by the domain partitioning are not independent. (b) One solution for recovering independent sets of connectors.*



On the left side of the figure, is an original partitioning of a 9-point grid into 4 subdomains. The middle lines overlap and the center point is assigned to all 4 subdomains. The initial keys associated with this mapping are shown above each node. This example shows that level-2 connectors defined directly from the domain degree $|Key(u)|$ may be adjacent. There are several possible remedies to change the keys so that the new connectors become independent. The (b) side of the figure shows one possible and natural solution which consists of expanding $Key(u)$ so that it contains the list of all the domains of each member of $adj(u)$, i.e., the set of all subdomains of the nearest neighbors of u. As will be seen in Section 2.2 this is not necessarily the best solution.

The original decomposition of a graph sometimes yield a hierarchical decomposition such that the connectors of a same level are uncoupled: this was the case for the 5-point mesh example seen earlier. By this we mean that in such situations, sets of connectors of the same degree will constitute a level for the hierarchical decomposition. Unfortunately, this situation is special. For an arbitrary irregular graph, the connectors obtained in this manner will not form a hierarchical interface decomposition in general. Indeed, connectors of the same degree can be adjacent. In the remainder of this section, we define first what properties we want to obtain from the interface decomposition and then we discuss algorithms to modify any interface decomposition in order to achieve these properties.

We need a property of a connector decomposition which will be called *consistency*. It is given next.

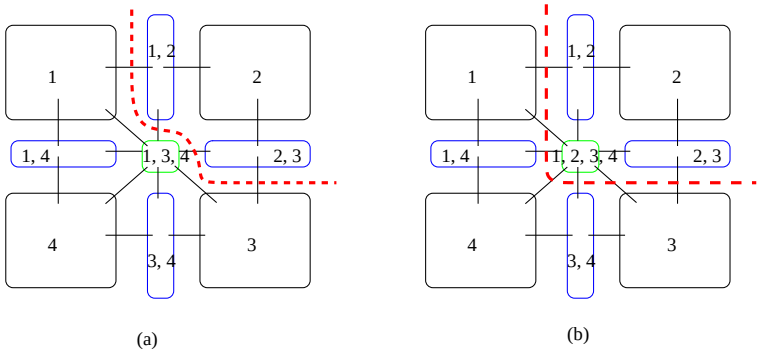
DEFINITION 2.1. *An interface decomposition is said to be consistent if any pair of connectors C_1, C_2 verifies:*

- *C_1 is adjacent to $C_2 \Rightarrow Key(C_1) \subsetneq Key(C_2)$ or $Key(C_2) \subsetneq Key(C_1)$.*

Another way to state the definition is that when two connectors are adjacent in a consistent interface decomposition then one connector key must be a strict superset of the other. In particular, two adjacent connectors must have different degrees, and the key of the one with higher degree contains (strictly) the key of the other. Figure 2.6 shows an example of a consistent and a non consistent interface decomposition. In case (a) the connector $C_{1,2}$ is adjacent to $C_{1,2,3}$ and yet none of the keys $\{1, 2\}$ and $\{1, 3, 4\}$ is included in the other. Similarly, $C_{2,3}$ is adjacent to $C_{1,3,4}$, yet $\{2, 3\} \not\subseteq \{1, 3, 4\}$ and $\{1, 3, 4\} \not\subseteq \{2, 3\}$. Modifying the

key of a connector is equivalent to modifying the overlap between subdomains. Thus, the dashed line in Figure 2.6 shows the boundary of subdomain 2 (overlap included). Note that subdomain k is the union of all connectors whose key contains the index k. In case (a) the connector $C_{1,3,4}$ is not a part of subdomain 2. whereas in case (b) the modified connector $C_{1,2,3,4}$ becomes a part of subdomain (2) as shown. The modification consisted of adding label (2) to the key of the connector $C_{1,3,4}$.

FIG. 2.6. *(a) Non consistent interface decomposition. (b) Consistent interface decomposition.*



An interface decomposition which satisfies the consistency requirement obeys two important properties which can be helpful in a parallel block ILU factorization.

LEMMA 2.1. *In a consistent interface decomposition, two connectors of the same degree cannot be adjacent.*
The proof of lemma 2.1 was essentially stated above, namely, two adjacent connectors have different degrees.

This property allows an independent elimination of all connectors of the same degree in a block ILU(0) factorization. Assume that a Gaussian elimination process is undertaken whereby the unknowns are ordered by blocks corresponding to the connector partition of the graph. The connectors, i.e., the blocks of unknowns, are ordered by increasing degree. Recall that in an ILU factorization based on the level-of-fill [15] there exists a fill-in entry of level k if and only if there exists a fill-path of length $k+1$ in the adjacency graph of the matrix. A fill-path is a path between two vertices v_i and v_j such that all intermediate vertices in the path have a lower label than those of v_i and v_j. Consider now a *block* ILU(k) factorization in which the blocks correspond to a partitioning of the unknown according to the interface decomposition. In the block factorization, fill-in is allowed in a given block by just using the level-of-fill pattern obtained from the quotient graph corresponding to the connectors. In this situation, blocks that are initially zero in the matrix A, and where fill-in will occur in the block ILU(k) factorization, correspond to fill-paths of length $\leq k+1$ in this quotient graph, which we call the connector graph. Note that in this graph, an edge between two connectors represents the adjacency between these connectors. A "fill-path" of length k is then a path of connectors $C_1, ..., C_k$ in the connector graph $\mathcal{C}$ such that

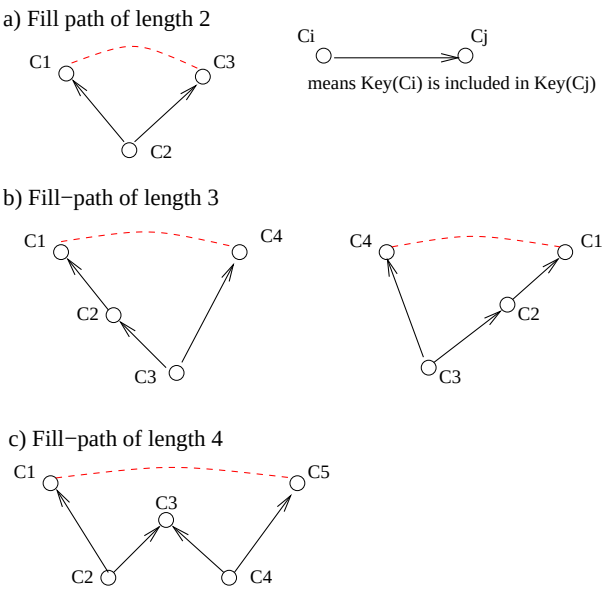
$$\forall i, 1 < i < k, deg(C_i) < deg(C_1) \;\; and \;\; deg(C_i) < deg(C_k)$$

since all connectors are numbered by ascending degree.

LEMMA 2.2. *Consider the connector graph $\mathcal{C}$ associated with a consistent interface decomposition. Then for a "fill-path" of length ≤ 3 there is always at least one domain number that is listed in all the keys of the connectors of the path.*

FIG. 2.7. *Proof of Lemma 2.2*

The cases a) and b) show all possible fill-paths of length ≤ 3 in the connector quotient graph. In these cases, the connectors in the fill path belong to at least a domain listed in the "minimum" connector key (the intersection of all keys of the connectors) of the path.
Thus, in case a) we have $Key(C_2) \subsetneq Key(C_1)$ and $Key(C_2) \subsetneq Key(C_3)$. In this situation $Key(C_1) \bigcap Key(C_2) \bigcap Key(C_3) = Key(C_2)$. In the case b), we also have $Key(C_1) \bigcap Key(C_2) \bigcap Key(C_3) \bigcap Key(C_4) = Key(C_3)$. This is true for the scenarios at the left and right of the case figure.
Case c) illustrates the fact that for a longer path we cannot say anything since $\bigcap_{i=1}^{5} Key(C_i) = Key(C_2) \bigcap Key(C_4)$, which may be empty. This is a situation where there is no common intersection (minimum) to all connectors: there are two local minima.



The proof is given in Figure 2.7 with some illustrative examples. A corollary of lemma 2.2, is that the fill-in induced by a block ILU(k) factorization with $k \leq 2$ of a matrix reordered and partitioned using a consistent interface decomposition is always contained in the local matrix of a subdomain. Section 3 will reconsider this in detail.

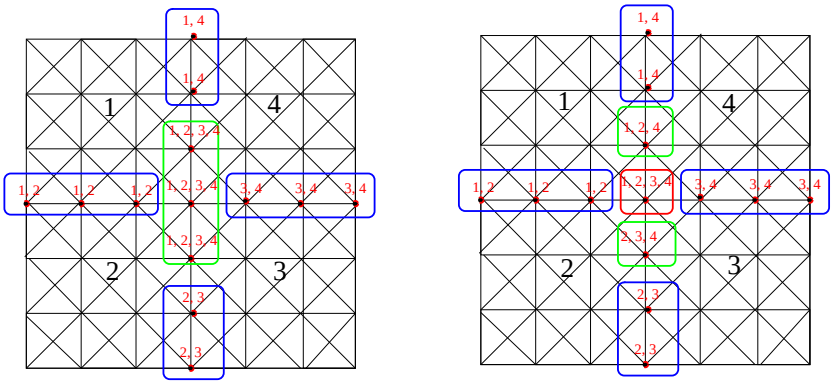
2.2. Algorithms for obtaining consistent decompositions. In the remainder of this section, we will discuss first an algorithm for obtaining a consistent decomposition of the interface into connectors. Then an algorithm to obtain levels of independent connectors will be discussed.

Figure 2.5-(a) shows that level-2 connectors defined directly from the domain degree $|Key(u)|$ may be adjacent. One possible remedy in this case, is to define $Key(u)$ differently. For example, we could define $Key(u)$ as containing the list of all the domains of each member of $adj(u)$, i.e., the set of all subdomains of the nearest neighbors of u. This fixes the problem but leads to interfaces that are unnecessarily 'thick' in some cases. Thus, in the example of the 9-point matrix of Figure 2.5-(a), the domain interfaces will be 3-line wide instead of just one line-wide. In this case, there is a remedy, which is to define $Key(u)$ in this manner only for non-interior nodes. This will give the solution shown in Figure 2.5-(b). A more general solution to get a consistent connector set is to start with a connector decomposition based on the initial overlap given by the partition and then refine it by *modifying, in a certain way, the keys* $Key(u)$ *of some overlapped nodes at the interface of the subdomains.* Thus, the solution in Figure 2.5-(b) consists of modifying 4 keys in the manner just described. Figure 2.8 shows two additional solutions to get independent connectors in a level, based on modifying keys.

Several possibilities arise in redefining keys to comply with the requirements of a hierarchical interface decomposition and some solutions appear to be better than others. In Figure 2.8 the solution (c) produces 3 levels (including the first level of interior node) in the overlap of the interface whereas solution (d) produces four levels. In (b) 4 keys are modified, resulting in 3 levels (degrees 1, 2, and 4), while in (c) 2 keys are modified resulting again in 3 levels, and in (d) 2 keys are modified resulting in 4 levels. Note that case (d) is not a consistent connector decomposition (Key $\{3, 4\}$ is not included in key $\{1, 2, 4\}$ but the corresponding connectors are adjacent). This example illustrates the fact that some post-processing of the keys are more desirable than others.

In the previous examples, we discussed two important criteria to use in irregular graph to find a good interface partitioning. The first is to limit the number of vertices whose key must be expanded to obtain a consistent connector set. The second one is to obtain as small a number of levels as possible. Thus, *the ideal interface decomposition should yield a small number of levels by modifying as few keys as possible.* In the general case, the problem of finding the best hierarchical decomposition of an interface is NP-complete

FIG. 2.8. *Two additional possible solutions to get independent connectors in each level by modifying some vertex keys.*



because the problem of finding the minimum vertex set to be modified in order to get independent level connectors is similar to a Minimun Set Cover problem [1]. Therefore, to define the connectors and levels as described before, we use two heuristics. The first algorithm aims at modifying the minimum number of vertex keys to enforce the consistency requirement. The second algorithm aims at building larger level sets in order to minimize the number of levels.

The principle of the first algorithm, Algorithm 2.1, is to "expand" iteratively some vertex keys of the same length, in order to uncouple connectors with the same degree. Note that expanding a vertex key means that this vertex will be overlapping additional subdomains than those given by the initial partitioning. In this algorithm we use the following notation:

- p is the number of subdomains;
- L^l is the set of vertices whose keys have a cardinality equal to l (i.e. vertices of domain-degree l);
- $V^l(i)$ is the set of vertices belonging to L^l and adjacent to vertex i;
- $Kdeg_u$ is the number of neighbors in $V^l(u)$ that have a key different from $Key(u)$.

ALGORITHM 2.1. *Computation of Independent Connectors*

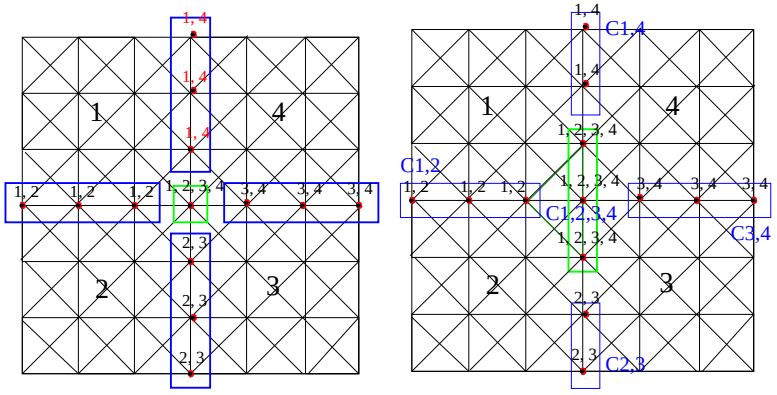
Initialization:
  For each vertex $u \in \mathcal{V}$
    $Key(u) :=$ *list of subdomains containing vertex* u.
  end
  For each vertex $u \in \mathcal{V}$
    $Kdeg_u := |\{\ v \in V^l(u)\ /\ Key(v) \neq Key(u)\}|$.
  end
  For $l = 1$ *to* p *Do:*
    $L^l = \{u \in \mathcal{V}\ /\ |Key(u)| == l\}$
  end
Main loop:
  For $l = 2$ *to* p *Do:*
    For each vertex $u \in L^l$ *do*
      $Key(u) := Key(u) \cup \bigcup_{k=1,\dots,l-1} \left(\bigcup_{v \in V^k(u)} Key(v)\right)$
    end
    While all vertices in L^l *have not been processed*
      Get u *the vertex in* L^l *such that* $Kdeg_u$ *is maximum.*
      $Key(u) := Key(u) \cup \bigcup_{v \in V^l(u)} Key(v)$
      $m = |Key(u)|$
      if $m > l$ *then*

21. $L^l := L^l \setminus \{u\}$
22. $L^m := L^m \cup \{u\}$
23. For each $v \in V^l(u)$
24. $Kdeg_v := Kdeg_v - 1$
25. end
26. end
27. end
28. end

At the initialization of Algorithm 2.1, the key of each vertex u is set to the list of subdomains that contain vertex u. In the main loop, the keys of each vertex u of degree 2 that is connected to another vertex of degree 2 with a different key is expanded by the union of all the keys from vertices in $V^l(u)$. Then the vertex u is set in a higher degree connector. This is continued iteratively over all the sets of connectors that have the same degree. Lines 13 – 15 of Algorithm 2.1 ensure the property of consistency (Definition 2.1). In this algorithm, the traversal order of the vertices of same degree is important since two different traversals can give very different results with a different number of vertices moved to an upper level. This heuristic finds a good traversal of the vertices of the same degree in order to modify as few vertex keys as possible to enforce the consistency condition. The principle is to define $Kdeg(u)$ as the number of neighbors in $V^l(u)$ which have a key different from $Key(u)$ and then choose the next vertex u in the traversal as the one that has the maximum $Kdeg(u)$ (line 17). For this we need to update on the fly the $Kdeg$ of the neighbors when a vertex expands its key (line 23,24,25). An implementation strategy is to sort the vertices by increasing order of $Kdeg$ in a max heap; the cost to obtain the maximum vertex is then constant and the cost to update the rank of a vertex neighbor is bounded by $O(log(n_i))$ where n_i is the number of vertices in the interfaces. If we denote by d_{max} the maximum degree of a vertex in the interface subgraph, then a pessimistic bound on the complexity of the main loop of the algorithm 2.1 is in $O(n + d_{max}.n_i.log(n_i))$. The initialization requires a loop over all the vertices to decide which ones are on an interface, thus it has a linear complexity in respect to the total number of vertices.

At the end of the algorithm, the key of a vertex u indicates the connector with which it has been associated, i.e., $C_{i_1,...,i_m} = \{u \in \mathcal{V} \mid Key(u) = \{i_1, ..., i_m\}\}$. Figure 2.9 illustrates the results of this algorithm on the 9-points grid example.

FIG. 2.9. *Partition of a 9-points grid into 4 subdomains. The connectors and values of the keys are shown at the beginning and at completion of the algorithm.*



2.3. The levelization algorithm. We have seen earlier that defining the level label of a connector as equal to its degree may yield a poor interface decomposition because an irregular domain partition may produce a large number of levels. A better definition of the levels from the keys is required to alleviate this problem. This definition requires an order relation $\overset{c}{>}$ for pairs of adjacent connectors.

DEFINITION 2.2. *For two adjacent connectors C_1 and C_2:*

$$C_1 \overset{c}{>} C_2 \quad \longleftrightarrow \quad Key(C_2) \subsetneq Key(C_1)$$

Note that for consistent decompositions, two adjacent connectors are always comparable with the relation $\overset{c}{>}$. The idea of the level computation algorithm is to compute iteratively the level as the set of local minimum for $\overset{c}{>}$ in the set of all unleveled connectors.

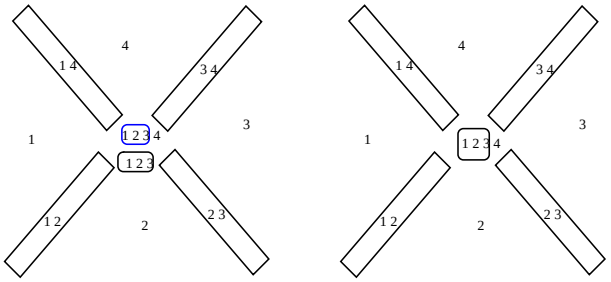
On irregular graphs, the connector decomposition can produce connectors that are of no benefit to the main goal of separating the graph into levels of independent sets. For example, Figure 2.10 shows a decomposition which produces two connectors $C_{1,2,3}$ and $C_{1,2,3,4}$ which should clearly be merged into one. Indeed, it is meaningless to differentiate between these two connectors because connector $C_{1,2,3,4}$ does not separate two connectors of the level immediately above it. A connector such as $C_{1,2,3,4}$ in Figure 2.10 will be termed a *singular connector*. A singular connector is always merged with a connector of a level immediately below it. The merge operation takes place just after a new level has been computed. The principle, described in Algorithm 2.3, is to merge any neighbor of the current level connectors such that the new merged connector is still a local minimum for $\overset{c}{>}$ in the set of all unleveled connectors.

In the algorithms 2.2 and 2.3, $\mathcal{C}_{unleveled.}$ is the set of all un-leveled connectors, $\mathcal{C}^i$ is the set of level i connectors.

ALGORITHM 2.2. *Level Computation Algorithm.*

1. $i := 0$;
2. *While* $\mathcal{C}_{unleveled.} \neq \emptyset$
3. $\quad i := i + 1$
4. $\quad \mathcal{C}^i :=$ *the set of all minimum connectors in* $\mathcal{C}_{unleveled.}$ *for* $\overset{c}{>}$.
5. $\quad \mathcal{C}_{unleveled.} := \mathcal{C}_{unleveled.} \setminus \mathcal{C}^i$
6. $\quad$ *Merge all the singular connectors for level* i.
7. *end*

FIG. 2.10. *On the left, connector* $C_{1,2,3,4}$ *is singular. On the right the singular connector has been merged.*



ALGORITHM 2.3. *Merge singular connectors for level* i.

1. *For each connector* $C \in \mathcal{C}^i$ *Do:*
2. $\quad$ *For each connector* $V \in \mathcal{V}(C) \bigcap \mathcal{C}_{unleveled.}$
3. $\quad\quad$ *if* $C \cup V$ *is a minimum for* $\overset{c}{>}$ *in* $\mathcal{C}_{unleveled.} \cup \mathcal{C}^i$ *then*
4. $\quad\quad\quad C = C \cup V$
5. $\quad\quad\quad Key(C) = Key(C) \cup Key(V)$
6. $\quad\quad\quad \mathcal{C}_{unleveled.} = \mathcal{C}_{unleveled.} \setminus V$
7. $\quad\quad$ *End*
8. $\quad$ *End*
9. *End*

One can notice that algorithm 2.2 will always produce a number of levels that is less than or equal to the number of different degree in the connector set. If we denote by D_{max} the maximum degree of a connector in the connector graph, by l the number of levels and by c the number of connectors then a pessimistic bound of the complexity is in $O(l.c.D_{max})$.

3. Parallel incomplete factorizations based on hierarchical interface decompositions. The previous section defined what we termed a "hierarchical interface decomposition" related to a partitioning

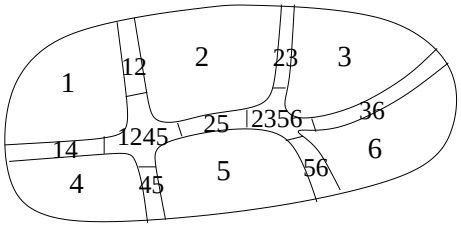
of the graph $\mathcal{G}$. This hierarchical interface decomposition consists of several levels of connectors which have the property of forming an independent set at each level. If the global matrix is (implicitly) reordered by level, starting with the nodes of level 1, followed by those of level 2, and so on, the independent sets will lead naturally to parallel ILU-type factorizations if careful dropping is used. This technique bears some similarity with the Algebraic Recursive Multilevel Solver (ARMS), discussed in [17, 9] with a major difference being that in ARMS the independent sets are determined dynamically whereas in the new algorithm, the ordering is set in advance. In addition, dropping is especially adapted to the partitioning as is explained in the next section.

3.1. Consistent dropping strategies. The interface decomposition and the keys associated with each node will be intimately related with the Incomplete LU factorization process. In particular, this decomposition and the keys will help define a suitable fill-pattern. The goal is to retain as much parallelism as possible in the ILU factorization.

The first principle of the elimination is to proceed by levels: nodes in the connectors of level 0 are eliminated first, followed by those in connectors of level 1, etc. All nodes of the zero-th level can be eliminated independently since there is no fill-in between nodes i and j of two different connectors of level 0 (interior of the subdomain). In contrast, fill-ins may appear between connectors from level 1 down to the last level of the interface connectors. These fill-ins will cause some uncoupled connectors to become coupled, and so they will no longer form an independent set. This will result in reduced parallelism.

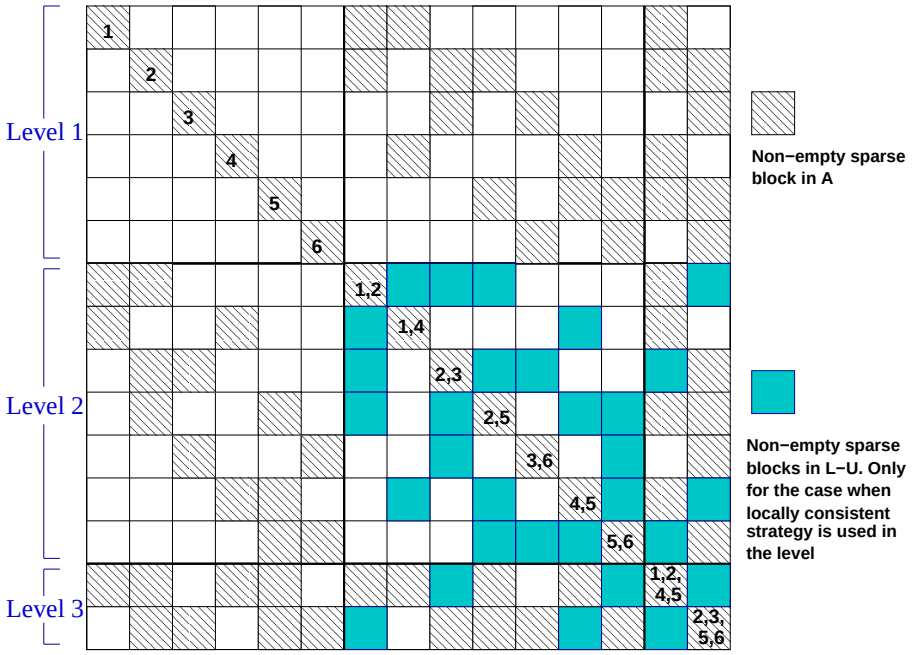
Recall that in the standard static ILU factorization a nonzero pattern is selected in advance based on the sparsity pattern of A. For example, the nonzero pattern for the ILU(0) factorization is simply the set of all pairs (i, j) such $a_{ij} \neq 0$, i.e., it is equal to the nonzero pattern of the original matrix. For details and extensions, see a standard text on iterative methods, e.g., [15, 2]. As was mentioned above certain nonzero patterns will allow more parallelism than others during the factorization. Certain choices of nonzero patterns may also lead to simpler implementations.

FIG. 3.1. *Domain partition into 6 subdomain. Connectors and their keys are represented*



We consider two possible options. The first is not to allow any fill-ins between two uncoupled connectors of the same level. In this fashion, the elimination of a any connector at a certain level can always proceed in parallel. The second option, which is less restrictive, is to allow fill-ins but to restrict them to occur only when i, j are associated to a common subdomain. In this case, elimination should be done in a certain order and parallel execution can be maintained by exploiting independent sets. Figure 3.2 draws the block structure of the global matrix reordered according to the Hierarchical Interface Decomposition (HID) of the domain illustrated in Figure 3.1. Figure 3.2 illustrates the two dropping strategies.

FIG. 3.2. *Block decomposition of the matrix according to the HID reordering. For convenience, all blocks have been drawn at the same size; using a real scale the blocks become significantly smaller in the high levels.*



3.2. Strictly consistent nonzero patterns. A nonzero pattern P is said to be strictly consistent with the HID if it does not allow any fill-in between any two connectors of the same level. This implies that an entry (i, j) will be in the nonzero pattern P only if vertices i and j belong to two adjacent connectors. In regard to the consistency requirement, this means:

$$P \subset \{(i,j) \mid Key(i) \subset Key(j) \text{ or } Key(i) \subset Key(j)\} \ .$$

In a block representation of the matrix, where blocking is done according to the levels, this means that this approach would discard all fill-ins between blocks, thus preserving the initial global block structure. A typical way to define strictly consistent nonzero patterns is to take a common dropping strategy and restrict the dropping further by imposing the above rule.

An important consequence of the restriction embodied in the above definition is that all connectors of the same level will form an independent set throughout the elimination process. Indeed, any fill-in in the diagonal block submatrices (corresponding to connectors that have the same level label) is discarded and thus the diagonal block structure is kept. In particular all eliminations will proceed in parallel at all stages. There remains only to consider how to handle the rows resulting from the elimination in different processors when these rows share two or more subdomains. Since these nodes are represented in more than one subdomain, they will undergo different modifications in each subdomain because the elimination performed in each subdomain is incomplete. In order to complete the process, we have two options. If i is the vertex index, we can have each processor in $Key(i)$ perform the same operations. Communication is required locally so that all processors will get the modifying rows. The second option is to select one processor in $Key(i)$ to perform all modifications. Once the modifications are completed, this processor will then send the final row to its neighbors. In the algorithms to be described later, this option is retained.

3.3. Locally consistent nonzero pattern. This type of nonzero patterns is less restrictive and allows *fill-ins* only between connectors belonging to the same subdomain. In other words, a fill-in is allowed in location (i, j) if $Key(i) \cap Key(j) \neq \emptyset$, as is stated in the following definition.

DEFINITION 3.1. *The nonzero pattern P is said to be locally consistent with the hierarchical interface decomposition if*

$$P \subset \{(i,j) \mid Key(i) \cap Key(j) \ \neq \ \emptyset\} \ .$$

Thus, an entry in location (i, j) of L or U is permitted only of $Key(i)$ and $Key(j)$ have at least one common index, i.e., only when i and j share one common subdomain. This permits an easy characterization of fill-ins based on the interface decomposition. Again, one can select a fill-in pattern such as ILU(k) or a dynamic one such as ILUT – with an additional constraint that no fill-in is allowed in locations not satisfying the compatibility property defined above. For the special case of ILU(k) with $k \leq 2$, it is interesting to note that no additional dropping would be exercised for the locally consistent dropping pattern, for any number of domains. This property is a direct consequence of Lemma 2.2. since a scalar ILU(k) non zero pattern is always contained in the block ILU(k) non-zero pattern.

In this situation, elimination can proceed simultaneously for nodes of two connectors of the same level whose keys are disjoint. As mentioned above, Gaussian elimination will eliminate the unknowns of the connectors level by level. A good way to exploit parallelism, is to partition the set of connectors in the level into maximum weighted independents sets. Here, the independent sets are for a graph whose vertices are connectors (identified by a key K) and the edges represent the binary relation $Key_1 \cap Key_2 \neq \emptyset$.

DEFINITION 3.2. *A set of connectors is "independent" if any two connectors of the set have disjoint keys. Such a set is "maximal" if it cannot be augmented by other connectors.*

Note that a maximal independent set is not one which has a maximum cardinality among all possible independent sets. The term "block independent sets" or "group independent sets" was used in [17, 9] for sets of subsets of vertices which are uncoupled. The idea of using independent sets of nodes (or subsets of nodes) in parallel Gaussian elimination is not new. Multicoloring was employed for example to define parallel ILU factorizations [7, 15]. Essentially, a global order – called "schedule" in [7] – in which to process the connectors is required. This global order can be defined by processing all those connectors which will not be coupled by any fill-in.

Figure 3.3 illustrates the elimination order that can be obtained in our incomplete factorization when a locally consistent strategy is used in all levels, on a simple example. With a strictly consistent strategy there is no need to find any special elimination connector ordering in a level since they can be eliminated independently.

3.4. Ordering of the unknowns inside the connector subsets. In addition to the global ordering of the connectors, a local ordering can be applied within each connector. In our codes, we apply a reordering of the interior domain unknowns (i.e., nodes within each level-1 connectors are reordered) to minimize fill-in for direct factorization. Such reorderings are provided by Scotch or METIS. No special reordering is applied to the higher level connectors.

3.5. Interpretation and generalizations. An interesting observation regarding the consistent dropping strategies just defined is that it can be viewed from the angle of quotient graphs. Define the quotient graph of connectors as a graph whose vertices represent the connectors and the edges the adjacency relationship between them. The dropping strategy can be viewed as a two level strategy which considers dropping within each connector independently from the dropping strategy between connectors. The former uses the original graph whereas the latter uses both the original graph and the connector graph.

In particular, notice that because the partitioning considered is edge-based (i.e., an edge never straddles two different subdomains), an edge in the connector graph between two connectors C_1 and C_2 implies that the two connectors are in the same subdomain. Consider now a zero fill strategy *on the connector graph*. In this strategy no edge will be created during the elimination process. Of course the ordering of the connector graph is consistent with that of the reordered matrix. An illustration is shown in Figure 3.4. In the case of strictly consistent dropping strategy, the block fill-in pattern is the same as that implied by an ILU(0) pattern on the connector quotient graph. Fill-in is obviously allowed within each connector. Because connectors that are adjacent in the quotient graph belong to the same domain, the second strategy defined earlier corresponds to allowing fill-ins of level ≤ 2 in the quotient graph (see lemma 2.2) . Generalizations can now be made. On can imagine applying a block ILU(k) with $k > 2$ on the connector quotient graph in the same way that this is defined in ILU(k) factorizations. Of course such dropping rules will entail more communication and a more complicated processing; for instance fill-ins may appear between connectors which do not share a common subdomain.

4. Parallel factorization. The partition of the unknowns into connectors induces a block structure of the matrix which enables an easy expression of the parallel algorithm. Figure 3.5 shows one such structure. The figure zooms in on the block-matrix associated with subdomain 2. The diagonal blocks correspond to all

FIG. 3.3. *Elimination of the connectors. The dense connector are the maximum independent set selected for the elimination. The dashed connectors are those already eliminated.*

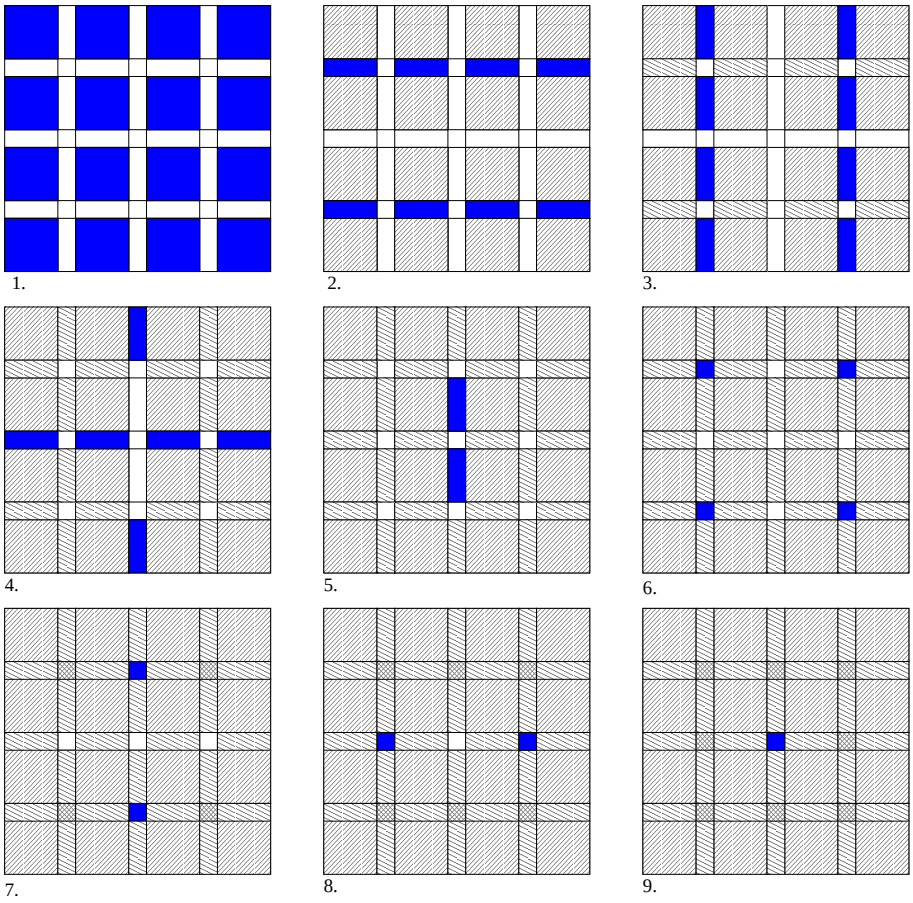
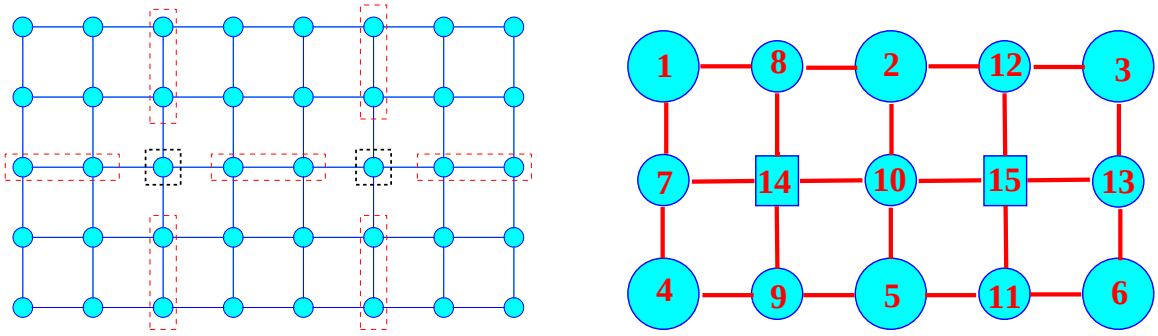
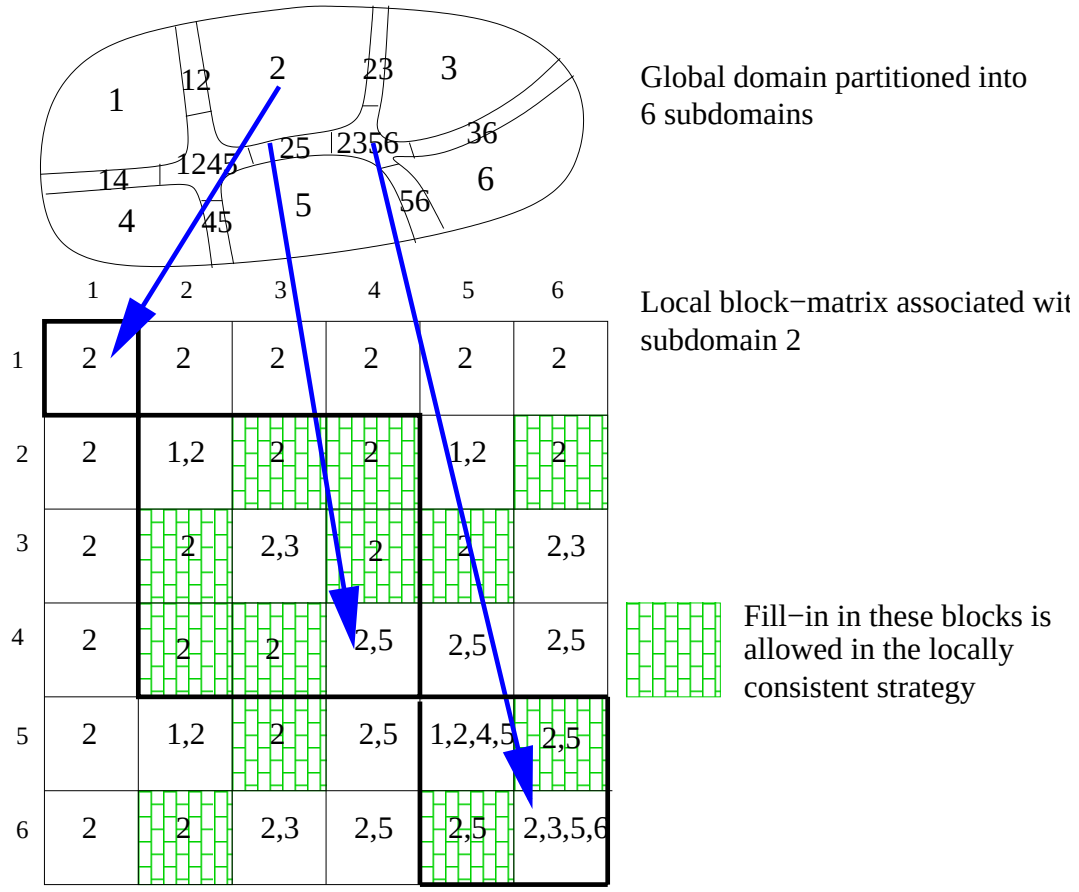


FIG. 3.4. *Hierarchical decomposition and corresponding quotient graph. Large circles show level-1 sets, small circles level-2 sets, and squares level-3 sets*



the connectors which are part of domain 2 and the off-diagonal blocks correspond to the coupling between these connectors. If each subdomain k is assigned to processor k, then the labels on each block is a list of all the processors which own a copy of the block. For example, the sub-block in position (4,6) of the block matrix, is labeled with 2 and 5, and therefore it is located in both processors 2 and processor 5. The label of a diagonal block is the key of the corresponding connector as defined previously. The label of an off-diagonal block (i, j) in the local block-matrix is the intersection of the labels of the blocks (i, i) and (j, j). For convenience, in the rest of this paper we will denote by $Key(i, j)$ the label of the block (i, j). Therefore,

FIG. 3.5. *Block partition of a local matrix*



using previous notation,

$$Key(i,j) = Key(i) \cap Key(j) \ .$$

The parallel factorization algorithm is given below (Algorithm 4.1). The following symbols are used in the algorithm.

- E represents the local matrix associated with processor p,
- C represents a local matrix where a block $C_{i,j}$ is a temporary location to add up all the updates to the block $E_{i,j}$. This matrix is empty at the start of the algorithm,
- `leader`$(p_1, .., p_m)$ is a function which selects a processor from a processor subset (see below for details),
- P is the non-zero block pattern of the matrix.

The function 'leader' is used to choose a unique processor to process a block matrix operation when several choices of processor are possible. In other word, this function is used to balance the workload and storage related to blocks overlapped by several local matrices on different processors.

Note that because of the ordering into levels, there is no explicit distinction between the levels in the algorithm. This distinction is inherent to the algorithm and is implicitly defined by the keys.

ALGORITHM 4.1. **Parallel Elimination Algorithm.**

0. Let N = block dimension of local matrix, and p = label of this processor.
1. For $i := 1$ to N
2. If $p = leader(Key(i,i))$
3. Receive and add in $E_{i,i}$ all $C_{i,i}$ from processors $\in Key(i,i)$

4. *Factorize* $E_{i,i}$ *into* $L_i.U_i$
5. *Send* $\{L_i, U_i\}$ *to processors* $\in Key(i,i) \setminus p$
6. *Else*
7. *Receive* L_i, U_i *from leader*$(Key(i,j))$
8. *EndIf*
9. *For* $j := i+1$ *to* N, *such that* $(i,j) \in P$ *and* $p = leader(Key(i,j))$
10. *Receive and add in* $E_{i,j}, E_{j,i}$ *all* $C_{i,j}, C_{j,i}$ *from processors* $\in Key(i,j)$
11. $E_{i,j} := L_i^{-1}.E_{i,j}$
12. $E_{j,i} := E_{j,i}.U_i^{-1}$
13. *Send* $\{E_{i,j}, E_{j,i}\}$ *to processors* $\in Key(i,j) \setminus p$
14. *EndFor*
15. *For* $j := i+1$ *to* N, *such that* $(i,j) \in P$ *and* $p \neq leader(Key(i,j))$
16. *Receive* $\{E_{i,j}, E_{j,i}\}$ *from leader*$(Key(i,j))$
17. *EndFor*
18. *For* $j := i+1$ *to* N, *such that* $(j,i) \in P$
19. *For* $k := j$ *to* N, *such that* $(i,k) \in P$
20. *If* $p = leader(Key(j,i) \cap Key(i,k)))$
21. $C_{j,k} = C_{j,k} - E_{j,i}.E_{i,k}$
22. *If* $j \neq k$
23. $C_{k,j} = C_{k,j} - E_{k,i}.E_{i,j}$
24. *EndIf*
25. *If all local contributions have been added up in* $C_{j,k}$
26. *Send* $\{C_{j,k}, C_{k,j}\}$ *to processors* $\in Key(j,k) \setminus p$
27. *EndIf*
28. *EndIf*
29. *EndFor*
30. *EndFor*
31. *EndFor*

A few additional details on the implementation will now be given. First, it is important to note that the forward and backward solves are also parallel at each level. The outer (level) loop is sequential. This is due to the structure of the diagonal blocks associated with the levels. Second, no pivoting is performed - but the standard forms of pivoting in the context of LU can be applied. For example, 'static pivoting' can be performed whereby diagonal entries are increased in an attempt to prevent the need for pivoting. Alternatively, we can also perform local block pivoting, whereby the pivot row is confined to stay within the dense block.

5. The preconditioning operation. The solution process consists in using the incomplete factorization as a right preconditioner combined with an iterative accelerator such as GMRES. The current version of PHIDAL, includes two ways of using the incomplete factorization. If we partition the matrix A according to an ordering associated with listing interior points first, followed by interface points, then we can define a block incomplete LU factorization as follows:

$$A = \begin{pmatrix} B & F \\ E & C \end{pmatrix} \approx \begin{pmatrix} L & \\ EU^{-1} & L_S \end{pmatrix} \times \begin{pmatrix} U & L^{-1}F \\ & U_S \end{pmatrix} \equiv M$$

Here we assume that $B \approx LU$ is an incomplete factorization of B and $L_S U_S$ is some incomplete factorization of an approximation to the Schur complement matrix $S = C - (EU^{-1})(L^{-1}F)$. In practice, with the above factorization we only need to store the factors L, U, L_S, U_S and the original blocks E, F. Given a right-hand side $y = \begin{pmatrix} y_B \\ y_C \end{pmatrix}$, the solution $x = \begin{pmatrix} x_B \\ x_C \end{pmatrix}$ to the system $Mx = y$ can computed as follows:

$$\begin{cases} x_C = U_S^{-1} L_S^{-1} (y_C - EU^{-1} L^{-1} y_B) \\ x_B = U^{-1}(L^{-1} y_B - L^{-1} F x_C). \end{cases} \tag{5.1}$$

An alternative is to explicitly compute and store some approximations G and W to the matrices $L^{-1}F$ and EU^{-1}, respectively. In this case the solution can be obtained from

$$\begin{cases} x_C = U_S^{-1} L_S^{-1} (y_C - W L^{-1} y_B) \\ x_B = U^{-1}(L^{-1} y_B - G x_C). \end{cases} \tag{5.2}$$

We refer to this as the M_{GW} preconditioner. One advantage of using the first variant (without the approximations W, G) is that it does not require to store the submatrices G and W; These matrices are actually available during the factorization process, as they are needed to obtain S, so no additional computation is incurred in this approach. In this version the G, W matrices are discarded after they are used and no longer needed. Another advantage is that this approach will tend to be more accurate than one which uses the approximations W, G. Therefore, the preconditioner M is usually more efficient than M_{GW}. On the other hand, the use of the G-W approximation will be slightly more expensive in general. Indeed, the number of floating point operations required to apply the preconditioner M is:

$$ope_M \approx 2\left(\mathrm{nnz}(L) + \mathrm{nnz}(U)\right) + \mathrm{nnz}(L_S) + \mathrm{nnz}(U_S) + \mathrm{nnz}(E) + \mathrm{nnz}(F) \ .$$

In contrast, applying the preconditioner M_{GW} which uses the W, G approximations costs roughly,

$$ope_{M_{GW}} \approx \mathrm{nnz}(L) + \mathrm{nnz}(U) + \mathrm{nnz}(L_S) + \mathrm{nnz}(U_S) + \mathrm{nnz}(G) + \mathrm{nnz}(W) \ .$$

In the situation when the B block is large, then the M preconditioner will cost more to apply in general, depending on the sparseness of the matrices W, G. Note that similar choices and issues were also discussed in the context of the ARMS preconditioner, [17]. In the numerical tests presented in the next section, the option $KeepGW = 0$ refers to the preconditioner M, and $KeepGW = 1$ refers to the preconditioner M_{GW}.

6. Numerical tests. Algorithm 4.1 allows many possible variations depending, for example, on the version of ILU that is used, whether consistent or strictly consistent dropping is applied, and on the selection of the various threshold parameters for dropping small terms. The current version of PHIDAL is based on an ILUT factorization (a numerical threshold is used to drop small entries). One can use two threshold parameters in the ILUT factorization $thresh_L$ and $thresh_U$ corresponding to the damping criterion for the lower triangular part and the upper triangular part of the matrix. In all our tests we set the same numerical threshold $thresh$ for L and U. The accelerator was GMRES with a restart parameter of 60. In Section 5, we mentioned two variant for using the incomplete factorization as a preconditioner ($KeepGW = 1$ and $KeepGW = 0$). In all the tables the fill ratio is defined as the number of non-zeros stored in the preconditioner divided by the number of non-zeros in the original coefficient matrix A. Thus, the fill ratio is smaller with the option $KeepGW = 0$ than with $KeepGW = 1$.

The preconditioning strategies used in the following tests utilize a combination of strictly and locally consistent dropping. Indeed, it appears appealing to use locally consistent strategies in the first few levels only and the less accurate and costly strictly consistent strategy at other levels. In the experiments, the number "Locally consistent levels", refers to the number of the first few levels (in the interface) in which dropping is performed in a locally consistent way. Strictly consistent dropping is performed thereafter.

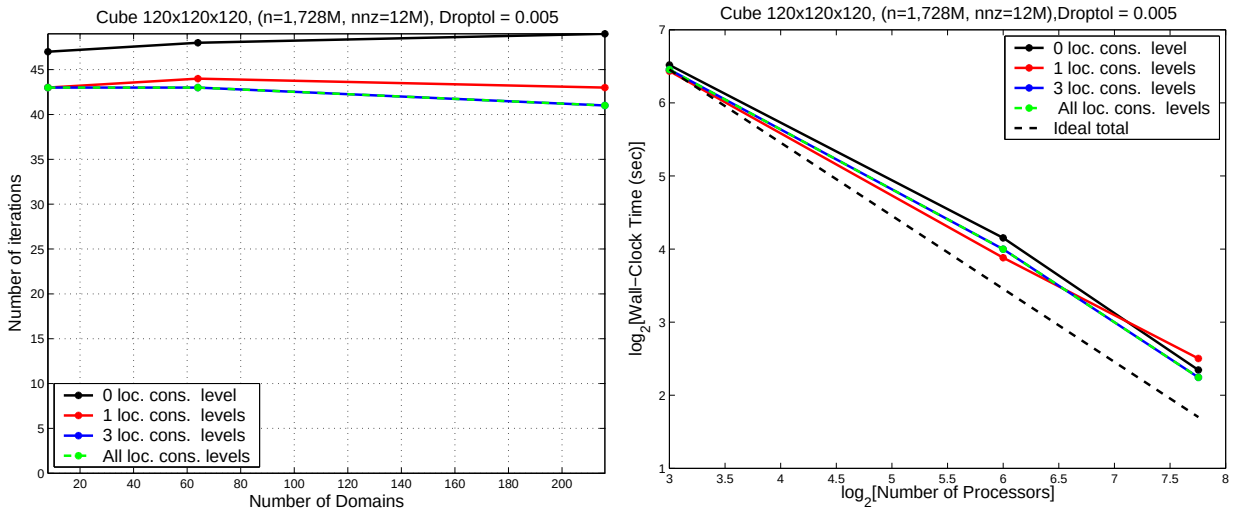
We used the graph partitioner METIS to get the initial domain partition and to reorder the interior domain unknowns in order to reduce fill-in. Similar results can be obtained by other graph partitioners and reordering algorithms to minimize fill-in.

6.1. Discretized 3-D Poisson equation. The problem solved is the elliptic partial equation $-\Delta.u = f$ on a square region with Dirichlet boundary conditions discretized with a seven-point centered finite-difference scheme on a $n_x \times n_y \times n_z$ (3D) grid, excluding boundary points. The mesh is mapped to a virtual 3-dimensional grid of $p_x \times p_y \times p_z$ processors. Therefore, subcubes of $r_x = n_x/p_x$, $r_y = n_y/p_y$, $r_z = n_z/p_z$, points, are mapped into each processor in the x, y, z, directions respectively.

The numerical dropping thresholds in the factorization were set to 0.01. With this drop tolerance the fill factor, i.e., the ratio of the memory required by the factorization over that for the original matrix, was about 4. The relative residual precision of the solution is 10^{-7}.

The machine used to run the tests was the IBM SP3 (named seaborg) of the NERSC, a computer equipped with NightHawks nodes clocked at 375Mhz. Each node has 16-processors which share a common pool of between 16 and 64 GB of shared memory.

FIG. 6.1. *Performance of GMRES accelerated with PHIDAL for* $120 \times 120 \times 120$ *Poisson equation as the number of processors varies.*



In this case, the decomposition in the interface is naturally defined by the partition (see Figure 2.1 in Section 2). The left part of Figure 6.1 suggests that the number of iterations required for convergence, is almost independent of the number of subdomains. The various cases shown correspond to the number of levels in the interface on which locally consistent dropping is performed. In the case of cube there are four levels of connectors (interior domain points, faces, edges and cross-points) and we can choose a locally consistent strategy for only three connector levels in the interface (faces, edges and cross-points). Thus the curves for *all locally consistent levels* corresponds to the case of 3 *locally consistent levels*. The right curves show that the time to precondition and to solve the problem scales very well with the number of processors; this shows that for this model problem, and for a good initial partition of the graph, the parallel factorization and the triangular solves are almost perfectly scalable in time.

We also ran an iso-scalability test on the same problem, where each processor was assigned a subdomain of size $40 \times 40 \times 40$. This means, for example, that on a virtual $3 \times 3 \times 3$ processor grid, the whole discretized problem is a $120 \times 120 \times 120$ mesh ($n \approx 1.7 * 10^6, nnz \approx 12 * 10^6$). Similarly, on a virtual $6 \times 6 \times 6$ processor grid the discretized problem corresponds to a $240 \times 240 \times 240$ mesh ($n = 13.8 * 10^6, nnz = 96.4 * 10^6$). Table 6.1 shows the iteration counts and the execution times for different numbers of processors. The numerical dropping thresholds in the factorization were set to 0.01; with this threshold the fill rate in the factors was about 4. These results were obtained with preconditioner M_{GW} (option $KeepGW = 1$).

TABLE 6.1
Scaled performance experiment: 3D grids $40 \times 40 \times 40$ *per processor, locally consistent strategy in all levels, numerical threshold set to* 0.01 *and* $KeepGW = 1$

$p_x \times p_y \times p_z$	Facto Time in sec	Solve Time in sec	Iter nbr	Fill ratio
3x3x3	6.10	28.85	52	3.98
4x4x4	6.58	42.05	63	3.98
5x5x5	6.80	57.60	78	3.99
6x6x6	6.81	66.53	84	3.99

6.2. Tests on a Magneto hydrodynamic problem. The matrices used in the following tests arise from a magneto-hydrodynamic flow problem that couples Maxwell's equation

$$\frac{\partial \mathbf{B}}{\partial t} - \nabla \times (\mathbf{u} \times \mathbf{B}) + \eta \, \nabla \times (\nabla \times \mathbf{B}) + \nabla \mathbf{q} = 0$$
$$\nabla \cdot \mathbf{B} = 0 \ ,$$

with the Navier-Stokes equation. Coupling is through the body force $\mathbf{f} = \frac{1}{\mu}(\nabla \times \mathbf{B}) \times \mathbf{B}$. In the following test, u is imposed, and Maxwell's equation solved for B. The finite element mesh is composed of mixed tetrahedra and hexahedra elements. Two matrices were used called MHD1 and MHD2, The matrix MHD1 is of size $n = 485,597$ and has $nnz = 24,233,141$ nonzeros. MHD2, which uses a different Reynolds number, is of size $n = 470,596$ and has $nnz = 23,784,208$ nonzero entries. For additional details on this problem, see also [16, 20].

The machine used to run these tests was a cluster of IBM power5 SMP nodes where each node is composed of 16 processors sharing a common pool of 32 GB of memory.

Levelization Algorithm. Figure 6.2 shows the algorithm of levelization in action for both MHD1 and MHD2 when the number of processors used is 256. The plots show the various levels obtained with the straightforward use of Algorithm 2.1 and then with the improved strategy based on Algorithm 2.2. For each of these levels the $\log_2$ of the number of nodes at that level is plotted. In the ideal situation we would have a small number of levels, or equivalently, we should obtain very few levels that have a small number of nodes as this hampers parallel execution. As can be seen, this is the case for both matrices. We should point out that for MHD2, the basic algorithm created 4 empty levels, i.e., levels with zero nodes, out of the 23 levels. These are the levels 18, 19, 20, and 22. Also, as expected, the effect of the levelization is not nearly as significant for fewer processors.

FIG. 6.2. *Effect of levelization algorithm. The plot shows the* $\log_2$ *of the number of nodes at each level obtained for a* 256 *domain partition. As desired, the levelization results in fewer levels (9 instead of 20), and, indirectly in far fewer small-size levels.*

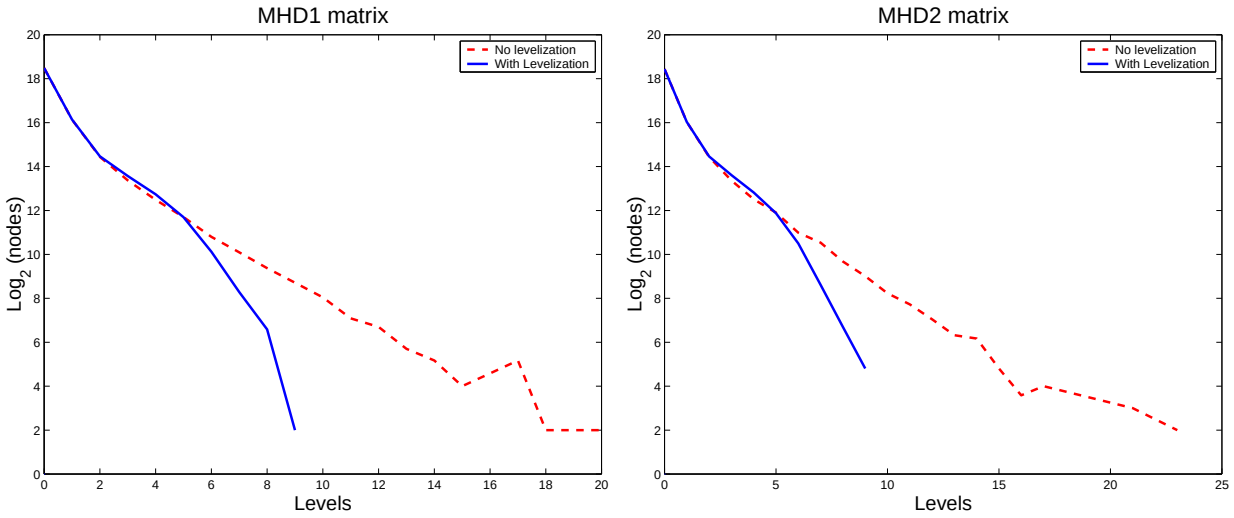


Table 6.2 gives the times (in seconds) to perform the hierarchical interface decomposition ordering for different number of domains on the MHD1 matrix. The first line indicates the percentage of nodes in the interface from the total number of nodes for the domain decomposition given in input. The second line gives the number of levels obtained before and after the levelization algorithm. The last line gives the time needed to perform the hierarchical interface decomposition ordering: it includes the times of the algorithm 2.1 (independent connector computation) and the time of the algorithm 2.2 and 2.3 (levelization algorithms).

We omit the details on the various times spent in each of these algorithms (we only report the total) because Algorithms 2.2 and 2.3 cost very little relatively to Algorithm 2.1. The fact that the time of the ordering for 32 domains is slighty smaller than for 16 processors is not significant and is certainly due to the accuracy of the timing.

The ordering time does not vary very much until the number of domains reaches 64. This is explained by the fact that in these cases a major part of the time is spent initializing Algorithm 2.1, consuming a time that is linear in the total number of nodes (interior domain and interface). When the number of domains increases the number of nodes in the interface grows and the time spent in the main loop of 2.1 grows, but the whole ordering time is still acceptable considering that the domain decomposition is usually done only once in a realistic physical simulation.

TABLE 6.2
Hierarchical decomposition for MHD1 n = 485597. The first line gives the number of domains. The second line gives the percentage of nodes in the interface between domains. The third line gives the number of connector levels obtained before/after the levelization algorithm. The fourth line gives the total time to compute the hierarchical interface decomposition.

Number of domains	16	32	64	128	256
Nodes in interface	9.1 %	12.3 %	15.4 %	20.1 %	24.4 %
Number of levels	6/5	10/6	11/8	15/9	20/9
Time (sec)	4.97	4.79	5.11	7.01	11.28

Effect of the type of dropping. The next experiments will show the effect of the dropping strategy on the quality of the preconditioner. Recall that we can define a strictly consistent scheme whereby no fill-in is allowed between connectors and locally consistent schemes whereby fill-ins are allowed between connectors at a given level, when these connectors are located in the same processor. We can also define an arbitrary number of levels that remain strictly consistent. Strictly consistent dropping allows to save memory, but as can be expected, sacrifices a little on the quality of the preconditioner. Clearly, the first level is always treated as a strictly consistent level. Beyond level 1, our code can decide to process as many levels in "locally" consistent mode as desired. It is important to process the last levels in strictly consistent mode since these will tend to generate a large amount of fill-in. Thus, a parameter is used to define how many levels from the second level down, are processed in locally consistent mode. The experiments below, show the executions obtained for 'zero' locally consistent levels (meaning that all levels are treated in strictly consistent mode), one locally consistent level (meaning that the second level is treated in locally consistent mode), and 'all' locally consistent mode. In the following experiments the threshold was always set to 0.002. The restart parameter in GMRES was again 60, and the stopping criterion is a reduction of the residual norm by a factor of 10^{-7}.

We reports the results with the option $KeepGW = 0$. The table 6.3 gives a comparison of the behavior of PHIDAL when the option $KeepGW$ is set to 1 (this table is to be compared with table 6.5).

TABLE 6.3
Left table is for MHD1 and right table is for MHD2.Number of locally consistent levels = 1, Thresh = 0.002, KeepGW = 1.

Proc	Facto (sec.)	Solve (sec.)	Total (sec)	Iter	Fill ratio
2	25.79	38.42	64.21	68	1.66
4	14.31	21.95	36.26	69	1.67
8	9.01	11.27	20.28	70	1.67
16	5.20	7.68	12.88	73	1.69
32	3.49	5.23	8.72	75	1.69
64	2.21	5.10	7.31	122	1.69
128	1.61	3.05	4.66	127	1.68

Proc	Facto (sec.)	Solve (sec.)	Total (sec)	Iter	Fill ratio
2	25.85	27.59	53.44	49	1.70
4	14.21	15.78	29.99	49	1.71
8	8.79	8.00	16.79	50	1.71
16	5.39	5.13	10.52	51	1.73
32	3.46	3.65	7.11	53	1.73
64	2.29	2.31	4.6	54	1.73
128	1.62	1.32	2.94	59	1.72

The main observation from the experiments is the remarkable behavior of the iteration number for the 'all' consistent levels, and to a lesser extent, for the one consistent level case. This iteration number remains about the same. On the other hand, this comes at a higher factorization cost, especially for the 'all' locally consistent case.

The compromise of using one locally consistent level seems a good one for this case. Note that this experiment suggests that using one or a few locally consistent levels would improve robustness without sacrificing performance too much.

On these problems, with higher numbers of locally consistent levels, the iteration counts remain small while the overall solution times as well as memory costs are higher. This trend is all the more important when the number of processors increases. It is difficult to set a general rule on the number of locally consistent levels to use, which may vary as a function of the desired precision, the matrix condition number and the connectivity of the adjacency graph of the matrix. Nevertheless, one locally consistent level should be a good choice in general since the bulk of the unknowns are usually interior points and points of the first level of connectors in the interface, whereas the higher connector levels contain few unknowns but require much more communication and synchronizations between processor when they are treated with a locally consistent strategy.

TABLE 6.4
Left table is for MHD1 and right table is for MHD2. Number of locally consistent levels $= 0$, *Thresh* $= 0.002$, $KeepGW = 0$.

Proc	Facto (sec.)	Solve (sec.)	Total (sec.)	Iter	Fill ratio
2	25.46	57.62	83.08	68	1.58
4	13.98	25.94	39.92	65	1.50
8	8.22	13.02	21.24	64	1.42
16	4.81	8.06	12.87	67	1.32
32	2.74	7.22	9.96	111	1.21
64	1.73	3.93	5.66	113	1.08
128	1.02	2.54	3.56	130	0.95

Proc	Facto (sec.)	Solve (sec.)	Total (sec.)	Iter	Fill ratio
2	25.90	39.68	65.58	47	1.62
4	14.14	19.41	33.55	45	1.53
8	8.56	9.32	17.88	46	1.47
16	4.83	5.12	9.95	45	1.34
32	2.79	3.19	5.98	51	1.22
64	1.76	1.78	3.54	54	1.11
128	1.01	1.14	2.15	64	0.96

TABLE 6.5
Left table is for MHD1 and right table is for MHD2. Number of locally consistent levels $= 1$, *Thresh* $= 0.002$, $KeepGW = 0$.

Proc	Facto (sec.)	Solve (sec.)	Total (sec)	Iter	Fill ratio
2	25.52	58.39	83.91	68	1.58
4	14.31	29.85	44.16	65	1.51
8	9.01	12.97	21.98	62	1.43
16	5.31	8.38	13.69	66	1.33
32	3.53	7.92	11.45	107	1.23
64	2.26	4.31	6.57	106	1.12
128	1.63	2.66	4.29	117	1.02

Proc	Facto (sec.)	Solve (sec.)	Total (sec)	Iter	Fill ratio
2	25.90	39.58	65.48	47	1.62
4	14.25	19.80	34.05	44	1.53
8	8.82	9.32	18.14	45	1.47
16	5.24	5.21	10.45	44	1.35
32	3.51	3.37	6.88	47	1.24
64	2.33	1.98	4.31	50	1.16
128	1.63	1.31	2.94	57	1.04

TABLE 6.6
Left table is for MHD1 and right table is for MHD2. All levels treated in locally consistent mode. Thresh $= 0.002$, $KeepGW = 0$.

Proc	Facto (sec.)	Solve (sec.)	Total (sec)	Iter	Fill ratio
2	25.54	57.68	83.22	68	1.58
4	14.34	29.30	43.64	65	1.51
8	9.23	12.89	22.12	61	1.43
16	5.60	8.66	14.26	66	1.34
32	5.18	8.55	13.73	106	1.24
64	4.68	5.59	10.27	100	1.14
128	4.95	3.90	8.85	107	1.06

Proc	Facto (sec.)	Solve (sec.)	Total (sec)	Iter	Fill ratio
2	25.90	39.64	65.54	47	1.62
4	14.25	19.51	33.76	44	1.53
8	9.20	9.83	19.03	44	1.47
16	5.59	5.37	10.96	43	1.36
32	4.98	3.68	8.66	45	1.25
64	5.27	2.55	7.82	46	1.17
128	5.34	1.88	7.22	50	1.08

Comparison with pARMS. We compared the performance of PHIDAL with that of pARMS on the MHD matrices discussed in earlier sections. These comparisons are not not easy to set up because pARMS uses different orderings which depend on numerical properties (see [9] for details) in the interior subdomain as

well as different rules for dropping small terms in the ILUT factorization. We used a set of parameters which seem to offer an acceptable comparison in term of fill-in ratio with PHIDAL. Because an iterative solver will usually require fewer iterations and less time if more fill-in is allowed in the preconditioner, an ideal comparison of quality would consist of testing both algorithm under the same (or close) memory requirements. Therefore, the parameters were set to achieve a fill-factor which is close for both methods. While pARMS allows the option of using an inner iteration for solving the Schur complement system of the interface, this feature is not yet available in PHIDAL. We use 5 inner iterations for pARMS, that is to say each time the preconditioner is applied, 5 inner GMRES iterations are invoked to solve the last Schur complement system, which is a much smaller system. The matrix graph was partitioned with METIS for both pARMS and PHIDAL.

The *thresh* parameter in ILUT was set to 0.002 for PHIDAL and pARMS. pARMS also uses a fill parameter that was set to 50. We used a relative residual tolerance of 10^{-6} for these tests. With this tolerance pARMS gives good results on both MHD1 and MHD2 with these parameters. For a tolerance of 10^{-7} pARMS does not convergence in less than 200 outer iteration for the MHD1 test case, though some other paramaters would have allowed to solve this problem. The results of PHIDAL are reported for the option $KeepGW = 0$ and all levels are treated in strictly consistent mode.

The comparison was limited to 64 processors because the pARMS test program which uses METIS as a graph partitioner requires that the whole matrix graph be loaded on one processor. The cluster on which the tests were performed does not offer sufficient memory per processor (when we ask for 128 processors) to run METIS on the whole matrix graph on one processor. Note that this is not a limitation of the pARMS library but a limitation due to the test program (provided in pARMS) which uses METIS as a graph partitioner. It is possible to use PARMETIS to circumvent the problem but this was not attempted.

TABLE 6.7
Test on MHD1. Left table is for PHIDAL and right table is for pARMS.

Proc	Facto (sec.)	Solve (sec.)	Total (sec)	Iter	Fill ratio
2	25.51	46.48	71.99	55	1.58
4	13.92	21.36	35.28	53	1.50
8	8.20	10.49	18.69	52	1.42
16	4.80	6.08	10.88	51	1.32
32	2.75	3.25	6.00	53	1.21
64	1.68	2.05	3.73	58	1.08

Proc	Facto (sec.)	Solve (sec.)	Total (sec)	Iter	Fill ratio
2	47.62	42.45	90.07	72	1.78
4	24.19	25.65	49.84	77	1.76
8	12.21	16.31	28.52	89	1.76
16	6.21	11.20	17.41	89	1.76
32	3.07	5.85	8.92	88	1.6
64	1.41	3.19	4.6	91	1.28

TABLE 6.8
Test on MHD2. Left table is for PHIDAL and right table is for pARMS.

Proc	Facto (sec.)	Solve (sec.)	Total (sec)	Iter	Fill ratio
2	26.13	33.20	59.33	40	1.62
4	14.05	15.40	29.45	38	1.53
8	8.42	7.84	16.26	39	1.47
16	4.60	4.41	9.01	39	1.34
32	2.79	2.63	5.42	44	1.22
64	1.74	1.49	3.23	46	1.11

Proc	Facto (sec.)	Solve (sec.)	Total (sec)	Iter	Fill ratio
2	44.83	33.21	78.04	57	1.78
4	23.95	19.52	43.47	61	1.76
8	11.90	11.05	22.95	63	1.76
16	5.61	7.55	13.16	63	1.6
32	2.81	4.39	7.2	64	1.6
64	1.28	2.13	3.41	63	1.92

TABLE 6.9
Test of pARMS with no inner iteration. Left table is for MHD1 and right table for MHD2.

Proc	Facto (sec.)	Solve (sec.)	Total (sec)	Iter	Fill ratio
2	47.89	46.09	93.98	87	1.78
4	24.92	25.96	50.88	92	1.76
8	12.29	15.97	28.26	110	1.76
16	6.22	9.74	15.96	113	1.76
32	3.07	5.40	8.47	116	1.6
64	1.41	3.43	4.84	131	1.28

Proc	Facto (sec.)	Solve (sec.)	Total (sec)	Iter	Fill ratio
2	44.88	38.40	83.28	73	1.78
4	23.96	21.82	45.78	81	1.76
8	12.30	12.55	24.85	85	1.76
16	5.62	7.90	13.52	89	1.6
32	2.81	4.04	6.85	97	1.6
64	1.28	2.18	3.46	102	1.92

Comparing the PHIDAL and pARMS total execution times shows a slight superiority of PHIDAL over pARMS for a small number of processors but the times become quite close when the number of processors increases. In both cases, the number of iterations is generally lower for PHIDAL than for pARMS. Since this comparison is only for two sample problems which are of the same type, it is not necessarily representative of a general situation. What can be said is that, overall, the performance is rather comparable for this problem and that more testing is required, specifically with a larger number of processors and larger problems, to reach a conclusion. Such an exhaustive comparison is beyond the scope of this paper.

7. Concluding remarks. We have presented a Schur-complement–type preconditioner which is designed to work for general sparse linear systems. The preconditioner is a multistage or multilevel technique which exploits a graph-based interface structure. Experiments show that the method scales quite well for model Laplacean problems, both in terms of time and iterations. Scalability was shown to be also quite good for general sparse systems arising from harder problems.

A number of improvements are still possible and are currently being investigated. The first concerns the inclusion of pivoting. Adding some form of local pivoting to PHIDAL is straightforward. A second issue is related to the graph partitioning. One may ask if it is possible to develop partitioning strategies that are especially tailored to enhance the overall performance of PHIDAL. In fact, partitionings that yield balanced graphs (with overlap) are still difficult to obtain in spite of the existence of excellent graph-partitioners such as METIS or Scotch. Finally, we plan to explore the impact of including an inner accelerator for solving the systems which arise at each level. This is a natural extension given the local matrix structure provided by the hierarchical interface decomposition.

Acknowledgment. The authors benefited from insightful discussions with John Wallis on the topic of hierarchical interface decomposition. In particular, the levelization algorithm was inspired from suggestions he made on reducing the number of levels and connectors. The authors also acknowledge Zhongze Li's extensive help in setting up the comparison with pARMS.

REFERENCES

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccameta, and M. Protasi. *Complexity and Approximation*. Springer-Verlag, 2003.
[2] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, New York, 1994.
[3] R. E. Bank and C. Wagner. Multilevel ILU decomposition. *Numerische Mathematik*, 82(4):543–576, 1999.
[4] E.F.F. Botta and F.W. Wubs. Matrix Renumbering ILU: an effective algebraic multilevel ILU. *SIAM Journal on Matrix Analysis and Applications*, 20:1007–1026, 1999.
[5] J. H. Bramble, J. E. Pasciak, and A. H. Schatz. The construction of preconditioners for elliptic problems by substructuring, I. *Mathematics of Computation*, 47(175):103–134, 1986.
[6] C. Farhat and F. X. Roux. Implicit parallel processing in structural mechanics. *Computational Mechanics Advances*, 2(1):1–124, 1994.
[7] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM Journal on Scientific Computing*, 22(6):2194–2215, 2001.
[8] George Karypis and Vipin Kumar. *MeTis: Unstrctured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0*, September 1998.
[9] Z. Li, Y. Saad, and M. Sosonkina. pARMS: a parallel version of the algebraic recursive multilevel solver. *Numerical Linear Algebra with Applications*, 10:485–509, 2003.
[10] M. Magolu monga Made and H.A. van der Vorst. A generalized domain decomposition paradigm for parallel incomplete LU factorization preconditionings. *Future Generation Computer Systems*, 17(8):925–932, 2001.

[11] M. Magolu monga Made and H.A. van der Vorst. Parallel incomplete factorizations with pseudo-overlapped subdomains. *Parallel Computing*, 27(8):989–1008, 2001.

[12] M. Magolu monga Made and H.A. van der Vorst. Spectral analysis of parallel incomplete factorizations with implicit pseudo-overlap. *Numerical Linear Algebra with Applications*, 9(1):45–64, 2002.

[13] F. Pellegrini. SCOTCH 4.0 User's guide. Technical Report, INRIA Futurs, April 2005. Available at URL `http://www.labri.fr/~pelegrin/papers/scotch_user4.0.ps.gz`.

[14] Y. Saad. ILUM: a multi-elimination ILU preconditioner for general sparse matrices. *SIAM Journal on Scientific Computing*, 17(4):830–847, 1996.

[15] Y. Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition*. SIAM, Philadelpha, PA, 2003.

[16] Y. Saad, A. Soulaimani, and R. Touihri. Variations on algebraic recursive multilevel solvers (ARMS) for the solution of CFD problems. *Applied Numerical Mathematics*, 51:305–327, 2004.

[17] Y. Saad and B. Suchomel. ARMS: An algebraic recursive multilevel solver for general sparse linear systems. *Numerical Linear Algebra with Applications*, 9, 2002.

[18] B. Smith, P. Bjørstad, and W. Gropp. *Domain decomposition: Parallel multilevel methods for elliptic partial differential equations*. Cambridge University Press, New-York, NY, 1996.

[19] B. F. Smith. *Domain Decomposition Algorithms for the Partial Differential Equations of Linear Elasticity*. PhD thesis, Courant Institute of Mathematical Sciences, September 1990. Tech. Rep. 517, Department of Computer Science, Courant Institute.

[20] A. Soulaimani, N. B. Salah, and Y. Saad. Enhanced GMRES acceleration techniques for some CFD problems. *Int. J. of CFD*, 16(1):1–20, 2002.