Parallel Direct Solution of Large Sparse Systems in Finite Element Computations

H. X. Lin
Department of Applied Mathematics and Informatics,
Delft University of Technology, Delft, the Netherlands

H. J. Sips
Department of Applied Physics, Delft University of Technology,
Delft, The Netherlands

Abstract

An integrated approach for the parallel solution of large sparse systems arisen in finite element computations is presented. The approach includes a three-phase preprocessor and a macro dataflow execution scheme. The three phases of the preprocessor are: (1) Extracting parallelism by means of an automatic domain decomposer; (2) Building the distributed data structure and (partial) scheduling for parallel computation during symbolic factorization; (3) Assigning processes (tasks) onto processors. The proposed approach has been implemented in the finite element analysis software package DIANA[1]. Experimental results show that this integrated approach is an efficient method for both shared- and distributed-memory parallel systems.

1. Introduction

The finite element method (FEM) is an important technique for the solution of a wide range of numerical simulation problems in science and engineering. Direct methods for the solution of a sparse system of equations play a prominent role in (commercial) general-purpose finite element analysis software packages (such as the DIANA package). A finite element computation in structural analysis using a direct method typically consists of a number of steps: input/mesh-generation, assembly of element stiffness matrices, ordering to determine the (node) elimination sequence, solution of the system of equations, and calculation of stresses with the computed displacement vector(s). Fig. 1 illustrates the procedures of a FE computation.

FEM applications are known to be very computation-intensive. The most computation-intensive part in a FEM computation is the solution of a large sparse system of equations. Therefore, the solution of a large system of equations must be parallelized in order to speed-up the FEM computation. DIANA is a large scale general-purpose FEM software package for structural analysis [1]. In this paper, we consider the matrix in the sparse system to be positive definite (the global stiffness matrix is generally positive definite in structural analysis applications).

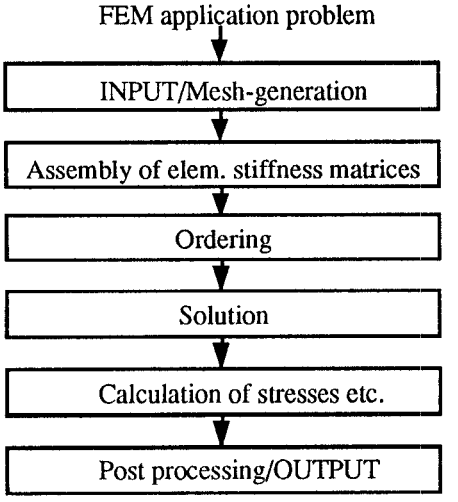


Fig.1 Illustration of the procedures of a FE computation for linear static analysis

Recently, a large number of studies on parallel equation solvers have appeared in literature (e.g. [2-7]). Many promising results on parallel iterative algorithms have been obtained. They demonstrate both a high achievable efficiency and scalability to a large number of processors. However, parallelizing direct method for large sparse systems turns out to be more difficult. Besides, when it comes to parallelizing an FEM software package (like DIANA), other parts of the computations such as assembly of element stiffness matrices, calculation of strains/stresses should also be parallelized in order to obtain a maximal benefit of the parallelization.

[1] DIANA is a registered trademark of TNO Institute for Building and Construction Research, Rijswijk, the Netherlands.

ICS-7/93 Tokyo, Japan
© 1993 ACM 0-89791-600-X/93/0007/0261...$1.50

In this paper, parallel direct solution of large sparse systems is considered. The presented approach is an integrated approach which starts with a domain decomposition. The resulting subdomains are assigned to the processors. In this way, the assembly of element stiffness matrices and the calculation of strains/stresses (after the solution of the sparse system) can be performed in parallel for each of the subdomains. Parallel computations of the element stiffness matrices and the calculation of stresses for the subdomains is quite straight forward, it can be done element by element and independent from each other. Therefore, we will concentrate on the parallel direct solution of sparse matrix systems in this paper.

A number of fundamental problems must be solved when solving a large sparse system of equations on a parallel and distributed computer:

1. Identifying parallelism in the problem;
2. Decomposing the computation into a number of subcomputations which can be computed concurrently;
3. Scheduling the subcomputations onto processors;
4. Distributing the data across the processors.

There are parallelizing compilers for shared-memory computers which tries to solve the first three fundamental problems automatically. Successes have been obtained for regular array operations. However, these vector and parallel compilers show performance for problems with irregular computation and/or data structures. E.g. when there are many gather and scatter operations (i.e. in direct memory addressing), current automatic vectorizers and parellelizers can hardly achieve any speed-up. In [8], results of experiment with the programs in the Perfect Benchmark problems [9] illustrate the state of achievement.

Because of its irregular computation and data structure of a Cholesky factorization for a sparse matrix, automatic vectorization compilers such as those on the current generation vector supercomputers are unable to achieve good speed-up. Automatic parallelization compilers for distributed-memory computers have been just recently put on the research agenda. For example, activities around High Performance Fortran [10] is an attempt to define a data parallel high level (array) language for parallel computers, to guide the distribution of data some compiler directives are provided. However, HPF and other array languages are so far data parallel oriented and support only regular (e.g. rectangular) array structures.

Dealing with a general-purpose software package such as the DIANA package adds an extra complexity to the parallelization problem. Parallelization by means of compilers has to solve the problem of decomposing a program into subcomputations during compile-time. However, the structure of a DIANA computation depends on the structure of the sparse matrix, and in turn the structure of the sparse matrix is dependent on the structure of the problem (the finite element domain). The structure of the sparse matrix can vary largely to a sky-crabber and when the element type changes from a linear triangle to a quadratic prism. Therefore, an optimal decomposition is only possible at run-time. Consequently, scheduling can not be done at compile-time, since it must follow the decomposition.

From the above discussions, we conclude that parallelization of direct solution of large sparse matrix systems in a FEM software package can not totally rely on advanced (future) parallel compiler techniques. Future development in parallel language and parallel compiler may ease the parallelization of our problem in consideration but special attention like the run-time analysis and run-time decomposition on each specific application problem will remain to be necessary.

2. A short review of previous approaches

In designing parallel algorithms, the key issue is inarguably the distribution of the workload among the processors. An optimal parallel algorithm usually requires to resolve the trade-off between a balanced distribution of workload and a minimal communication- and synchronization overhead. In the following, we will shortly review several known approaches to the problem of parallel factorization of large sparse matrices.

George et al [5] consider sparse Cholesky factorization on a distributed computer system. They describe a column-Cholesky factorization scheme, in which columns corresponding to the nodes at the same level of the elimination tree are eliminated independently. The granularity of the parallel algorithm is at the level of modifying (updating) a column. In their algorithm, a large communication overhead occurs due to the fine granularity of parallelization at the column level, although the results obtained on a hypercube simulator show the inter-processor communication has been quite evenly distributed among the processors. Also such a fine granularity incurs a large synchronization overhead. The effects of these two overheads are clearly reflected in the performance of their algorithm, the reported maximal speed-up is only 4.19 for 8 processors and 5.54 for 16 processors.

In [3] Duff et al have considered the implementation of the multi-frontal method on shared-memory processors. Good vectorization results have been reported in [7] for the multi-frontal code implemented with level 3 BLAS routines. The (multi-)frontal method has the advantage of requiring only a minimal core storage. But the assembly of frontal matrices and the factorization of them have to be intervened with each other, leading to a complicated computational structure. This drawback becomes more severe when we want to implement the method on a distributed system, since elements of a frontal matrix have to be transferred and redistributed before and after the assembly. The multi-frontal method is better suited for shared-memory (super)computers with a small number of (vector) processors, but less suited to distributed-memory architectures.

In [6] Gilbert and Schreiber consider fine-grain parallel factorization of large sparse matrices. In their scheme, the unit of operation to be distributed across the processors is a single multiplication or addition of two scalar numbers. This approach aims at maximally utilizing the parallelism inherent in the factorization process. However, their

experiments show that the obtained parallel efficiency is very low because of the large communication- and synchronization overhead incurred by this fine-grain approach. This result confirms with our analysis that medium- to large-grain approach is the best trade-off in dealing with parallel factorization of large sparse matrices. This is certainly the case for distributed parallel systems with a relative large communication and computation speed ratio (all current generation of parallel computers do have a large ratio).

3. Outline of the integrated approach for parallel finite element computations

Generally, the problem of parallelizing a complex software package like the DIANA package can be identified as the following: 1. Extraction of parallelism --- How to extract parallelism inherent in finite element computations; 2. Data structures --- What sort of data structures are appropriate for parallel (distributed) computation; 3. Assigning processes (tasks) onto processors --- How to assign tasks to the processors by resolving the trade-off between load-balance and communication- and synchronization overhead such that the parallel completion time is minimal.

When parallelizing a general-purpose FEM software package like DIANA the design criteria are considered to be crucial importance:

A. the parallel FEM package should be able to function without direct user intervention on matters with respect to parallelization. This is extremely important because the users of a FEM package (in our case mechanical engineers) are not interested in the underlying intricacies of (parallel) computer systems. This implies that any parallelization generated by the system must be *automatic* and *efficient*.
B. the resulting parallel software should be portable. For reasons of maintainability, the number of versions of the (parallel) solvers should be very limited. There should be only one basic version, which can be applied to sequential- or parallel computers.

With these premises in mind, a preprocessor has been implemented to free the users (e.g. mechanical engineers) from the burdens of parallelizing their FEM applications (remember that nowadays parallel programming is still a hard job even for an experienced computer programmer). The preprocessor consists of three parts: 1. An automatic domain decomposer; 2. Symbolic factorization and set up of concurrent data structures for parallel computation; 3. Heuristics for assigning tasks onto processors.

Analysis in [11] shows that for a domain consisting of n nodes linear speed-up can be achieved with this approach with upto $p=O(\sqrt{n})$ processors. For example, for an $\sqrt{n}$ by $\sqrt{n}$ square grid, the best sequential factorization time is known to be $O(n^{3/2})$ [12]. With this approach subdividing the domain into $\sqrt{n}$ subdomains, the parallel factorization time is $O(n^{3/2}/p)$, and the communication overhead is $O(p\cdot\sqrt{n})$. Therefore, the linear speed-up performance scales up to $O(\sqrt{n})$ processors.

4. Domain decomposition

In order to handle regular and irregular structured element domains with different type of elements, the domain is transformed into a connectivity graph and the presented domain decomposition algorithm is based on graph partitioning. The problem of partitioning a graph into a number of balanced (equal-sized) subgraphs with a minimum separator length between the subgraphs is known to be NP-complete [13]. So, heuristics have to be used for partitioning a graph.

Two of the most popular methods are the dissection methods [12] and the minimum-degree algorithms [14]. The dissection methods (one-way or nested) are simple and fast, but give a not very optimal separator length. An alternative is to use a minimum-degree algorithm to perform a (sequential) ordering first. Next, a partitioning is performed on the elimination tree corresponding to this ordering ([15], [16]). The method using a minimum-degree algorithm generally gives better results than the dissection methods in terms of separator length. However, for a graph with n nodes a minimum-degree algorithm has a time complexity of $O(n^2)$ as compared to that of $O(n)$ to $O(n.log(n))$ for the dissection methods. This means that the method using a minimum-degree algorithm has a higher time complexity as the solution of the sparse matrix system (e.g. $O(n^{3/2})$ for a $\sqrt{n}$ by $\sqrt{n}$ 2-D grid). Moreover, the graph partitioning algorithm is hardly vectorizable and difficult to get parallelized. Therefore, in order to avoid the domain decomposition becoming the most time consuming part of the whole computation, the graph partitioning algorithm must be relatively fast (e.g. with a time complexity less than $O(n.log(n))$).

The automatic domain decomposer described in [17] is similar to the one-way dissection method. It differs only at two points from the one-way dissection method. The domain decomposer in [17] always returns subdomains of the same size (with a difference of at most 1 element). The one-way dissection first constructs a level-structure [12] and cuts at a level resulting in a subdomain with size closest to the "required" size. This generally results in a smaller separator. Also, we have found through experiments that the algorithm in [17] tends to produce scattered subdomains (i.e. a subdomain consists of more than one separated subgraph component). This can however be fixed by changing the near-random "grasping" procedure of taking elements into a subdomain to an orderly one by grasping elements into a subdomain line-wise (surface-wise in 3-D case) instead of hopping from one element to another.

Many other domain decomposition or graph partitioning algorithms can be found in literature. One of the classical works in this field is the Kernighan-Lin algorithm [18]. The Kernighan-Lin algorithm begins with an initial partition of the graph into two subsets A' and B', which differ in their size by at most one. At each iteration, the algorithm chooses two subsets of equal size to swap between A and B, thereby reducing the number of edges

that join A to B. This algorithm has the so called "hill-upclimbing" ability to eventually reach the global optimum, however, the computational time complexity is very high (each iteration takes a time of $O(n^3)$). In [19], Chrisochoides et al present a domain decomposition algorithm which uses geometrical information (coordinates). The algorithm produces smaller separators than the Kernighan-Lin algorithm in some cases. However, its applicability is merely limited to regular grids with equidistant grid-points. Another method is the spectral partitioning algorithm [20]. It has been reported that it generates partitions comparable to algorithms like Nested Dissection and Kernighan-Lin. The time complexity for a bisection is $O(n^3)$. Recently, some researchers have considered using simulated annealing techniques for graph partitioning [21]. Since these methods are essentially combinatorial algorithms, their time complexity is very high. In some applications, when a partition is to be used many times (e.g. to analyze the structure under a variety of different loads), it might be profitable to spend more time on getting a better partition with one of these algorithms.

Our domain decomposition algorithm begins with an initial decomposition. This initial decomposition is determined according to a variant of the dissection method. Then in order to shorten the length of separators between the subdomains, the initial decomposition is improved by a bipartite maximum matching algorithm [22]. Measures are taken to adapt the iterative separator improvement algorithm for bisection to the general N-partition problem.

Fig. 2 shows the graph partitioning algorithm DD. In Fig. 2, the following parameters have been used: *NRSUBD* = number of subdomains to be partitioned; SUBNL = (total number of elements in domain) */NRSUBD*; *MINNL* = minimum number of elements per subdomain; and *MAXNL* = maximum number of elements per subdomain.

Algorithm DD (Domain Decomposition)

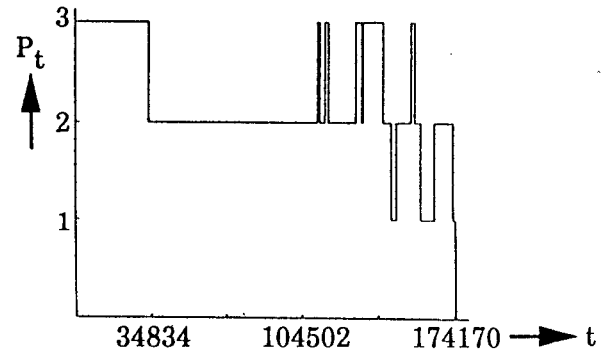
```
domain = initialize with all elements in the finite
         element domain;
ISUB = 1;
while ( domain is not empty ) do
   select an (peripheral) element L with minimum
   degree;
   S(-1) = { Φ }; S(0) = { L };
   i = 0;
   domain = domain\{L};
   while ( | |S(i)| - SUBNL | < | |S(i-1)| - SUBNL | and
         NOT stop ) do
      construct the (i+1)-th level S(i+1);
      if ( Σ_{K=0}^{i+1} |S(k)| < MAXNL ) then
         Sort S(i) into an ordered elements of
         line/surface;
         i = i+1;
      else
         stop = TRUE;
      endif;
   enddo;
   Adjust the ISUB-th subdomain's size; subject to the
   constraints of MINNL and MAXNL;
   call refinement procedure IMPROV0 (local);
   call refinement procedure IMPROV1 (non-local);
   domain = domain \ subdomain(ISUB);
   Adjust SUBNL; try to make the number of partitions
   as close to the original NRSUBD as possible, while
   satisfying the conditions MINNL≤SUBNL≤MAXNL;
   ISUB = ISUB+1;   /* next subdomain */
enddo;
```

Figure 2 A description of the domain decomposition algorithm (DD).

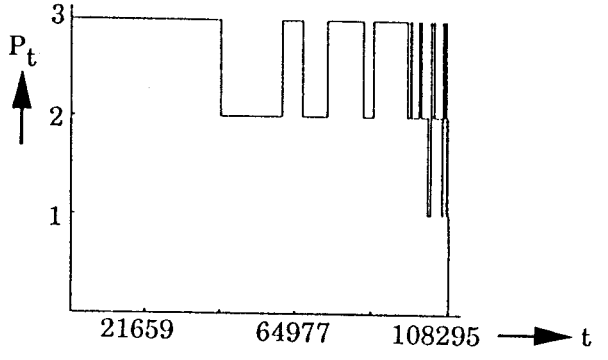
One important feature of our domain decomposition algorithm is that the optimal number of subdomains is chosen on the fly, depending on the expected ratio between computation- and communication speed, and the minimization of the number of fill-ins. In literature, a domain decomposition, if applied for a p processors system, always focus on partitioning the domain into p subdomains of the same size. The aim is to obtain a good load-balance among the p processors. While this is often adequate for a good load-balance for many iterative methods (for solving the sparse system of equations), our experimental results show that having p subdomains for p processors does not guarantee a good load-balance in solving the sparse system by a direct method. For instance, in a 2-D rectangular domain the workload corresponding to a subdomain in the corner or along the border can differ significantly from that of a subdomain in the centre. Fig. 3 illustrates the relationship between load-balance and the number of subdomains. So, for a good load-balance it is preferred to have $k \cdot p$ subdomains with k is an integer and larger than *1*. The optimal number of subdomains depends on the ratio between computation- and communication speed, and the minimization of the number of fill-in's. As the increase of the number of subdomains will increase the communication- and synchronization overhead, the break-even point is a function of the size of the element domain.

Another feature of our domain decomposition algorithm which differs from the most other domain decomposition algorithms is that the size of a subdomain may vary from a *min_size* to *max_size* and is determined in the course of the optimization. The requirements of equal sized subdomains and minimum separator length are two conflicting goals. An optimal decomposition requires to

resolve the trade-off between these two conflicting goals. To allow the size of a subdomain vary from *min_size* to *max_size* leaves more choices for the decomposition algorithm to minimize the separators. At the same time, the load-imbalance caused by this relaxation is limited, because now with $k.p$ ($k>1$) subdomains combinations can be made to correct the imbalance. This relaxation is especially effective in the case that a direct method is used to solve the sparse system of equations.



(a)

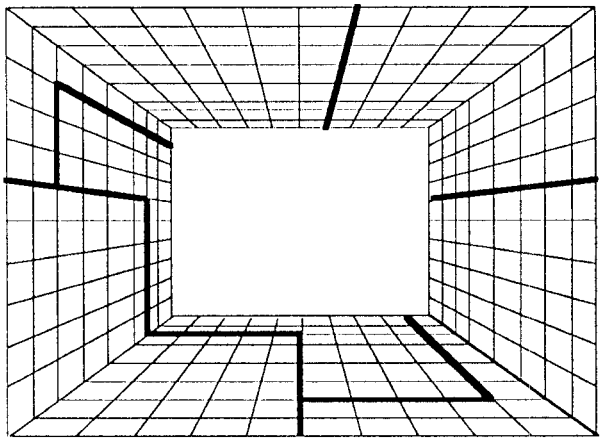


(b)

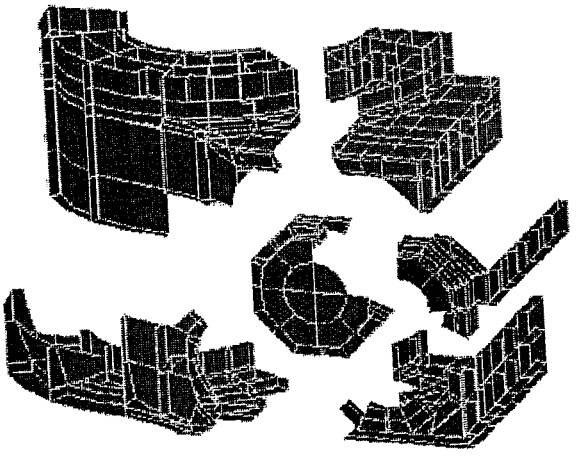
Fig. 3. (a). Execution profile of an example problem divided into 3 subdomains mapped onto 3 processors. P_t is the number of tasks which are executed in parallel at a time. (b) idem as a. now for 6 subdomains mapped onto 3 processors.

In algorithm DD, the procedures IMPROV0 and IMPROV1 are called. These are the procedures performing the "iterative" separator improvement based on the principle of maximal matching of a bipartite graph. Liu [22] has described a method for improving the separators of a bisection for a 2-D grid problem. In IMPROV0 this method is generalized to an N-partition problem. IMPROV1 uses the idea of maximal matching of a bipartite graph to obtain some global improvement of the separators [11].

The time complexity of our domain decomposer is $O(n)$ to $O(n.log(n))$. Fig.4 shows two examples of domain decomposition produced by the algorithm DD. Fig.4a represents a masonry wall subdivided into 5 subdomains, and Fig.4b represents a deceleration top (3-D) subdivided into 6 subdomains.



(a)



(b)

Figure 4. (a). A masonry wall subdivided into 5 subdomains; (b) A decelerating top (3-D) subdivided into 6 subdomains.

5. A block storage scheme for sparse matrices

An efficient data structure for sparse matrices should have a minimal storage requirement and at the same time support fast computation and communication in a parallel and vector computing environment. Three basic requirements on the data structures are:

(1). Efficiency --- the data structure should only store those entries which are logically nonzero in a sparse matrix;
(2). Distributable --- the data structure should support the distribution of parts of the sparse matrix in terms of block-matrices in a natural way;
(3). Fast access --- It should provide fast row-wise access to the block matrices. For example, the consecutive elements of a (sparse) row should be accessible consecutively

without performing indirect addressing. This will provide good vectorizability of the basic operations.

Fig.5a shows a problem domain decomposed into 4 subdomains. The nodes along the border between two subdomains are called interface nodes. If we order all the nodes in the subdomains before the interfaces, a doubly bordered block diagonal matrix (DBBD-matrix) results [11]. Fig. 5b illustrates the block data structure of the sparse matrix corresponding to a decomposition of a problem into four subdomains as shown in Fig. 5a. In Fig. 5b, a submatrix on the main diagonal ((1,1), (2,2), (3,3), (4,4)) corresponds to the stiffness matrix of the interior nodes of a subdomain and it has the structure of a profile matrix. A submatrix along the border ((5,1), (5,2), (6,2), etc.) corresponds to the interaction between an interface and a subdomain and is stored as a collection of sparse (row-)vectors. The submatrices in the right-bottom corner of the DBBD-matrix ((5,5),(6,5),(6,6), etc.) correspond to the interactions between the interfaces themselves, they are dense matrices. During the symbolic factorization phase, it is determined where the fill-in's will occur. From this information, the (static) data structure is built. A block-oriented data structure has the advantage of being well structured and well vectorizable. A basic set of submatrix-operations can be defined for this DBBD-matrix. These basic submatrix-operations can be seen as a sparse variant of the level 3 BLAS routines [23]. The profile matrices and the border matrices stored as a collection of sparse vectors can be very well vectorized, this block data structure is thus suited for both vector- and parallel processing.

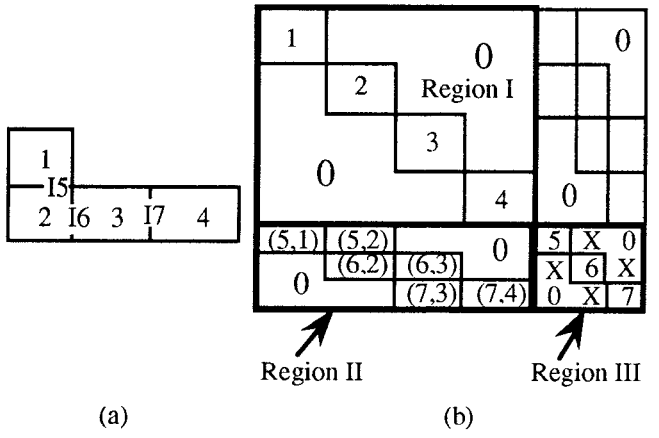


Figure 5. (a). A problem domain decomposed into 4 subdomains. (b). The structure of the corresponding DBBD-matrix. An "X" represents a block fill-in.

6. Ordering for parallel factorization and minimum fill-ins

The solution of a large sparse matrix system with the direct method usually can be divided into the following phases: 1. Ordering; 2. Symbolic factorization; 3. Numerical factorization; 4. Triangular solution (forward- and backward substitution). In sequential computation, ordering considers the elimination sequence of the nodes in order to minimize the number of fill-ins and the number of arithmetic operations. In parallel computation, ordering should consider the parallel elimination sequence of nodes as well. This means that a trade-off between the number of fill-ins (therefore the number of operations) and parallelism in the elimination sequence has to be resolved.

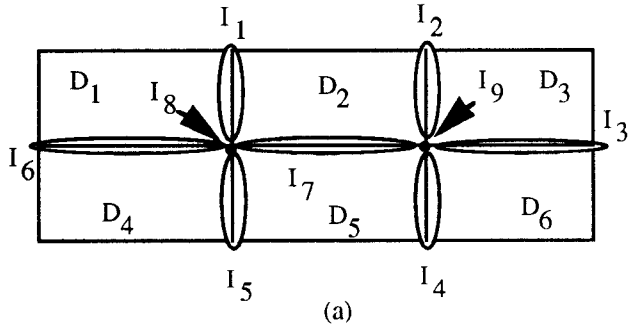
As to the parallel elimination sequence in case of domain decomposition, each of the subdomains (the submatrices D_j) is assigned to one processor and the factorization of D_j can be computed independently from each other[2]. The submatrices in Region II can be computed parallelly once the corresponding D_j at the same column has been factorized. Update conflicts may occur when updating (e.g. the Schur-complement) the submatrices in Region III. In this section, we consider the sequence of factorization of the diagonal submatrices, and leave the sequence of updates to be determined in a later stage. The ordering is thus a *partial scheduling*. The subdomains (i.e. submatrices in Region I) are to be eliminated first (and independently), so the problem left is how to determine the elimination sequence of the interfaces.

If we define an *interface segment* or simply an *interface* as a group of connected nodes adjacent to the same set of subdomains (see Fig. 6a), and if each of the subdomains and interfaces is considered as an aggregated node. Then a new graph can be defined with these aggregated nodes as nodes and the adjacency relation as edges, this is called a *quotient graph* (Fig. 6b). After the elimination of the subdomains the elimination graph with the interfaces as nodes results (Fig. 6c), ordering techniques for graphs with the original node in the finite element mesh can be applied for the quotient graph. Notice that with this definition of interfaces, the submatrices representing the interface-interface interactions (Region III) are dense matrices. This means that the storage scheme in Region III is very compact. All nonzero submatrices to be stored are dense matrices while the zero submatrices are not stored.

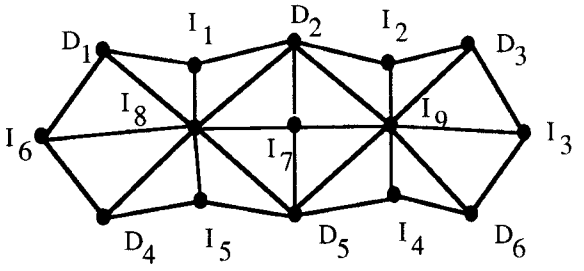
In graphical terminology the problem of partial scheduling can be described as follows. The ordering of the interfaces transforms a undirected interaction graph into a directed elimination graph (an interface node can be eliminated in the next step if all the nodes with outgoing edges to this node are already eliminated). The procedure of ordering is as follows. After domain decomposition, an interaction graph can be defined with the subdomains and interface segments as aggregated nodes and the adjacency among them are represented by edges. We know a priori (from the consideration of minimizing fillins) that the subdomains are eliminated before the interfaces (the subdomains are independent to each other so no ordering is required for them). After the elimination of the subdomains, fill-edges are added among the interfaces. This results in an interaction graph of the interfaces. Next, the

[2] Although the submatrix Dj can be parallel factorized using more than one processors, we consider here only the case that Dj is factorized sequentially.

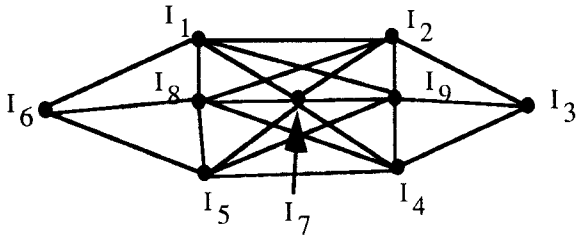
elimination sequence of the interfaces are determined. In the following, we will discuss techniques and algorithms for partial scheduling the elimination of interfaces.



(a)



(b)



(c)

Figure 6. a. Illustration of subdomains and interface segments; b. the quotient graph; c. the (quotient) elimination graph after eliminating the subdomains.

Minimum degree (MD) algorithm has been one of the most popular and commonly used technique for ordering to achieve minimum fill-ins (e.g. [14]). The MD-algorithm selects a node with the least number of edges connected to it as the pivot to be eliminated in the next step. The MD-algorithm can be adapted for ordering the (interface) quotient graph. In the following, we will describe two minimum degree approaches for ordering the interfaces.

A. The quotient graph as a uniweighted graph

In a uniweighted quotient graph, each node represents an interface segment. All the edges are considered to have the same unit weight. A MD-algorithm for ordering this uniweighted quotient graph is the same as the conventional MD-algorithm.

B. The quotient graph as a weighted graph

The nodes and edges in a weighted quotient graph are the same as in the uniweighted graph. The difference is that the edges are each assigned with a weight which may differ from 1. There are many possible ways to define the weight of an edge in a quotient graph. An appropriate definition for the weight of an edge between node i and j form the point of view of node *i* is *Card(j)* (*Card(j)* is the number of nodal nodes of interface *j*), and the degree for node *i* is

$\sum_{j \in Adj\{i\}} Card(j)$, where *Adj{i}* is the set of neighbours of interface node *i* in the elimination graph. The MD-algorithm selects the node with the minimum degree defined above.

Besides the MD-algorithms, an alternative is to use the local minimum fill algorithm. A local minimum fill algorithm selects the node, whose elimination causes the least fill-ins, as the pivot for elimination at the next step. The local minimum fill algorithm generally results in fewer fill-ins than the MD-algorithms (for details are referred to [11]). So far, the ordering algorithms considered are for minimum fill-ins. Next, we will consider the problem of introducing parallelism by ordering.

INPUT: G^1 = adjacency graph of interfaces after eliminating the subdomains;

$S_1 = \Phi$; i=1;
while ($G^i \neq \Phi$) do
  done = false;
  determine the minimum degree MinDeg in G^i and mark all nodes V_{G^i} in G^i as schedulable;
  while (not done and G^i has at least 1 schedulable node) do
    find a schedulable node N with minimum degree in $G^{i'}$;
    if (Degree(N) $\leq$ [MinDeg($G^{i'}$) + α]) then
      add N to S_i;
      mark all neighbours of N in G^i as unschedulable;
      if (Card(S_i) = p) then
        done = true;
      endif;
    else
      done = true;
    endif;
  enddo;
  i = i + 1;
  $V_{G^i} = V_{G^{(i-1)}} \setminus S_i$;
  $E_{G^i} = E_{G^{(i-1)}} \setminus \{$all edges with an end node $\in S_{(i-1)}\}$
    $\cup$ { (n_1,n_2) | N$\in S_{(i-1)} \wedge n_1 \in$ Adj(N) $\wedge$
    $n_2 \in$ Adj(N) in $G^{(i-1)}$ };
enddo;

OUTPUT: the elimination sequence $S_1, S_2, \ldots, S_k$.

Figure 7. A controlled multiple MD-algorithm.

Two nonadjacent nodes in the elimination graph can be eliminated independently [11]. Fig. 7 shows a parallel elimination scheduling algorithm, it is called the *controlled multiple MD-algorithm*. A node N is selected for (parallel) elimination only if $Degree(N) \leq MinDeg(G^{i'}) + a$, where $G^{i'}$ is the remaining elimination graph, and a is an integer value. A larger value of a favours parallelism while a smaller value of a restricts parallelism in favor of minimizing fill-ins.

Once the elimination sequence $S_1, S_2, ..., S_k$ has been determined, the structure of fill-ins in the factor is determined. The relative sequence of interface nodes with the same set S_i only affects the relative positions of the fill-in submatrices but it does neither alter the number of fill-ins nor the parallelism. Therefore, the structure of the factor with fill-ins can now be determined. The determination of the structure of the factor is usually called the symbolic factorization.

7. A macro dataflow implementation

As the ordering is only a partial scheduling (it only determine the parallel elimination sequence of interfaces), the computation sequence of the updates in the Schur complement (Region III) needs yet to be determined. After the symbolic factorization the structure of the factor with fill-ins is known. Let S_0 be the set of subdomains, and S_t $(1 \leq t \leq k)$ the sets of interfaces which can be independently eliminated. Further let L the lower triangular factor, $C(j)$ the set of indices of nonzero submatrices at column j of $(L^T - I)$ (or equivalently row j of $(L-I)$). Then the parallel block LDL^T-factorization algorithm can be described as in Fig. 8. Notice that in Fig. 8 the product term $L_{i,k} \cdot D_{k,k}$ occurs many times in the computations. In the implementation we use temporary storage places to keep these product terms in order to minimize the number of operations required.

for t=0, 1, ..., k
  do parallel for i in S_t
    do parallel for j in C(i)

$$A_{i,j} = A_{i,j} - \sum_{k \in C(i) \cap C(j)} A_{i,k} \cdot L_{k,k}^{-1} \cdot L_{j,k}^{T}$$

$$L_{i,j} = A_{i,j} \cdot L_{j,j}^{-1} \cdot D_{j,j}^{-1}$$

    od;

$$A_{i,i} = A_{i,i} - \sum_{k \in C(i)} A_{i,k} \cdot L_{k,k}^{-1} \cdot L_{i,k}^{T}$$

    Factorize $A_{i,i} \rightarrow L_{i,i} \cdot D_{i,i} \cdot L_{i,i}^{T}$
  od;
od

Figure 8. A parallel LDL^T-factorization algorithm.

The computation structure of the parallel LDL^T-factorization algorithm can be represented as a task dependency graph. A (sub)matrix is considered as an integral object, and each computational task in de dependency graph relates to computing/updating one and only one object. The factorization of the submatrices corresponding to the subdomains can be performed as the first group of parallel tasks.

With the definition of the computational task the precedence relation between tasks can be transformed to data dependency relations between (data) objects. This data structure provides a base to implement a macro dataflow execution scheme. Fig. 9 shows a dependency graph for the DBBD-matrix in Fig. 7b. On shared-memory systems, the macro dataflow scheme is implemented by a ready-task queue, where all tasks ready for execution are placed. Whenever a processor becomes available, the first task is picked out of the queue and executed. Because a task only modifies one object (i.e. a block of data), a single semaphore is sufficient to resolve the access contention problem.

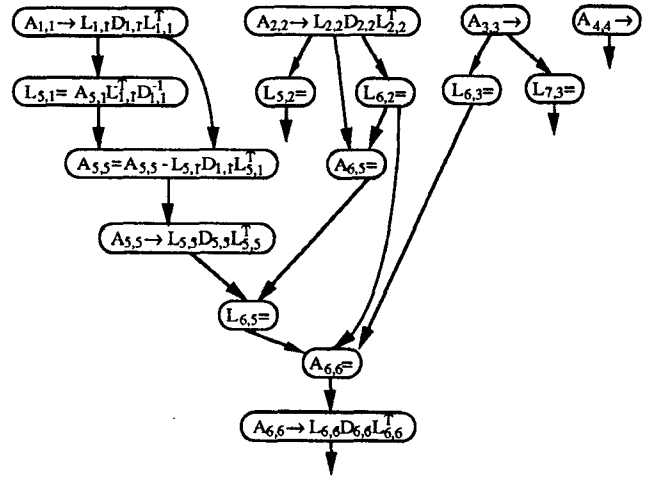


Figure 9. Illustration of the task dependency graph for the matrix in Figure 5b.

On distributed-memory systems, the access to a central ready-task queue can be a bottleneck, especially for massively distributed parallel computers. Moreover, since data are stored in local memories of the processors, a dynamic task assignment can drastically increase the amount of communications. Therefore, we have chosen the approach of assigning the computation to processors prior to the execution. Each processor now has its own ready-task queue to be accessed and updated. The parallel programs for both shared- and distributed-memory systems can be implemented in the SPMD (Single Program Multiple Data) paradigm [24]. The few differences are communication statements instead of lock/unlock of semaphores.

For task assignment, several schemes have been investigated [25]. A cyclic block column-wise (CBC) scheme assigns a block column to a single processor, and a cyclic block (CB) assignment scheme assigns the tasks in a task graph cyclically among the processors. The CBC

scheme can lead to a higher load imbalance than the CB scheme, however, from the communication point of view CBC is to be preferred above CB. In our current implementation, the submatrices on the chief diagonal and along the borders are assigned using the CBC scheme, while the submatrices in Region III (see Fig. 7b) are assigned using the CB scheme. This has been shown to be an efficient heuristic, as it can also be observed from the results in next section.

8. Some experimental results

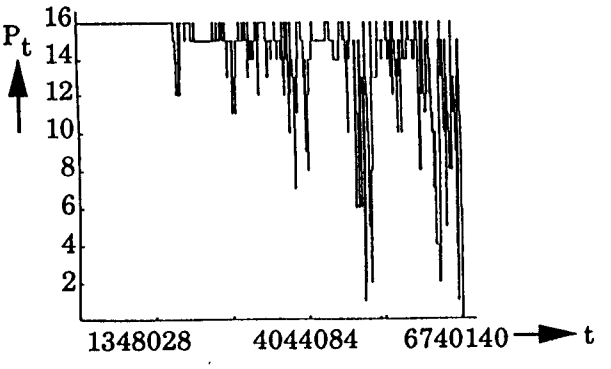
In Table I, the performance of the parallel direct solver (incl. factorization and solution of the triangular systems) on a Convex C240 is shown. Except the first two problems, the test problems are taken from real applications of DIANA users. (1) The problem Mesh 1 is a 2-D mesh consisting of 20 by 40 square elements; (2) Mesh 2 is a 2-D problem consisting of 30 by 60 quadratic square elements; (3) A building structure consisting of 567 quadratic square elements, 18 quadratic triangular elements, and 130 beam elements; (4) A 3-D structure modelling the joint between bricks, it consists of 1400 quadratic hexahedron elements; (5) A 3-D model of a half of a (symmetric) stopcock in a gas production installation, it consists of 388 quadratic hexahedron elements.

It can be observed that efficiency figures of higher than 90% have been obtained. It should also be mentioned that the parallel solver is faster than the old (sequential) frontal solver of DIANA when it runs on a single processor (e.g., the problem 'deceleration top' is solved twice as fast, and the large 3-D problem of brick-joint can not be solved with the old solver, but has been solved with 2 processors of the Convex as fast as the frontal solver running on a Cray X-MP (with 1 processor). This is merely due to the more efficient ordering and the efficient block data structure in the parallel solver.

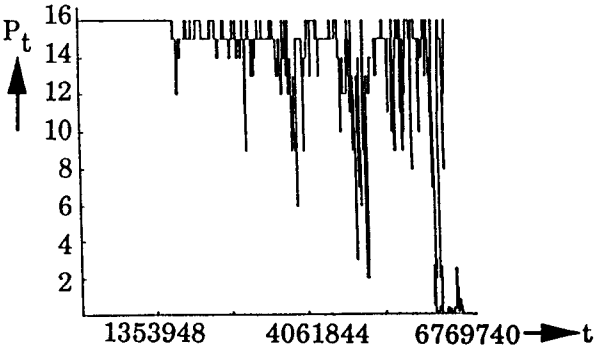
Table 1. Performance results of the parallel solver on a Convex C240.

Problem	p = 1	p = 2		p = 4	
	time (s)	time (s)	speed-up	time (s)	speed-up
Mesh 1 (2D) (5038 eqs)	10.13	5.33	1.90	3.02	3.35
Mesh 2 (2D) (11158 eqs)	22.19	11.63	1.91	5.95	3.73
Appartment (3916 eqs)	10.67	5.65	1.89	3.67	2.91
brick-joint (3D) (18883 eqs)	2166.82	1103.74	1.96	601.39	3.60
Stopcock (3D) (7425 eqs)	88.83	46.98	1.89	23.97	3.71

Fig. 10 shows the simulation results of a distributed-memory system with 16 processors. These results are obtained using a simulator which simulates the execution of a task graph. Fig. 10a shows an execution profile without communication overhead, and Fig. 10b shows the same task graph simulated with the iPSC/2 communication timing characteristics. Communication overhead typically results in a performance degradation of less than 5%. An efficiency higher than 85% has been obtained in Fig. 10. These results conform with the analysis in [18] that linear speed-up can be attained with a number of processors upto $p=O(\sqrt{n})$, where n is the number of equations. Currently, the finite element software package DIANA is being ported for distributed memory parallel computers.



(a)



(b)

Figure 10. Simulated execution profile of a task graph resulting from a decomposition of 32 subdomains.

9. Conclusions

An integrated approach for parallel direct solution of large sparse systems from the finite element computations has been presented. An automatic domain decomposer is used as a preprocessor for extracting parallelism right from the problem description (i.e. the element model). We show that to have p equal-sized subdomains does not guarantee balanced load distribution in case of direct solution. The presented domain decomposer takes this into account in the optimization procedure. To enable a structured and efficient data communication, a block data structure for the sparse

matrix has been proposed. Furthermore, a sophisticated ordering algorithm, the so-called controlled MD-algorithm, for (partial) scheduling for parallelism on the one hand and minimizing fill-ins on the other hand has been presented. All these, complemented with a macro dataflow execution scheme, provide an efficient and portable parallel program for finite element computations.

Acknowledgement

This research was supported in part by TNO Institute for Building and Construction Research and the Dutch Information Technology programme SPIN. The authors wish to thank Len Dekker for this research work. Arjan van Gemund for many useful discussions, and the DIANA group at TNO-BOUW Institute.

References

[1] R. de Borst, G.M.A. Kusters, P. Nauta and F.C. de Witte, "DIANA - a comprehensive, but flexible finite element system", In *Finite Element Systems: A Handbook*, Ed. C.A. Brebbia, Springer, Berlin, 1985.

[2] O. Storaasli, J. Ransom and R. Fulton, "Structural dynamic analysis on a parallel computer: the finite element machine", *25th AIAA/ASME/ASCE/AHS Structures, Structural Dynamics and Materials Conference*, Palm Springs, CA. May 14-16, 1984.

[3] I.S. Duff, "Parallel implementation of multifrontal schemes", *Parallel Computing* 3, 1986, pp. 193-204.

[4] C. Farhat and E. Wilson, "A new finite element concurrent computer program architecture", *Int'l J. for Num. Meth. in Eng.*, Vol. 24, 1987, pp. 1771-1792.

[5] A. George, M.T. Heath, J.W. Liu, and E. Ng, "Sparse Cholesky factorization on a local-memory multiprocessor", *SIAM J. Stat. Comput.* 9, 1988, 327-340.

[6] J.R. Gilbert and R. Schreiber, "Highly parallel sparse cholesky factorization", *SIAM J. Sci. Stat. Comput.*, Vol. 13, No. 5, 1992, pp. 1151-1172.

[7] P. Amestoy, M. Dayde and I. Duff, "Use of Level 3 BLAS kernels in the solution of full and sparse linear equations", Report TR 89/9, CERFACS, 1989.

[8] W. Blume and R. Eigenman, "Performance analysis of parallelizing compilers on the Perfect Benchmarks programs", *IEEE Trans. Parallel and Distributed Systems*, Vol. 3, no. 6, Nov. 1992.

[9] M. Berry, et al, "The perfect club benchmarks: Effective performance evaluation of supercomputers", *Int. J. Supercomput. Appl.*, Vol. 3, no. 3, pp. 5-40, Fall 1989.

[10] High Performance Fortran Forum, *High Performance Fortran Language Specification*, Version 1.0, Rice University, Houston, Texas, January 1993.

[11] H.X. Lin, *A Methodology for Parallel Direct Solution of Finite Element Systems*, PhD Thesis, Delft University of Technology (to appear).

[12] A. George, J.W. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, 1981

[13] M.R. Garey, D.S. Johnson, and L. Stockmeyer, "Some Simplified NP-complete graph problems", *Theoret. Comput. Sci.*, Vol. 1, 1976, pp. 237-267.

[14] A. George and J.W.H. Liu, "The evolution of the minimum degree ordering algorithm", SIAM Review, Vol. 31, No. 1., 1989, pp. 1-19.

[15] J.W.H. Liu, "The role of elimination trees in sparse factorization", *SIAM J. Matrix Anal. Appl.*, Vol. 11, no. 1, pp. 134-172, Jan. 1990.

[16] C.C. Ashcraft, "The domain/segment partition for the factorization of sparse symmetric positive definite matrices", Eng. Comp. & Anal. Tech. Report ECA-TR-148, Boeing Comp. Serv., November 1990.

[17] C. Farhat, "A simple and efficient automatic FEM domain decomposer", *Computers & Structures*, Vol. 28, No. 5, 1988, pp. 579-602.

[18] B.W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs", *The Bell System Tech. J.* 49, 1970, pp. 291-307.

[19] N.P. Chrisochoides, E.N. Houstis and C.E. Houstis, "Geometry based mapping strategies for PDE computations", *In Proceedings of ACM Supercomputing Conference '91*, 1991, pp. 115-127.

[20] A. Pothen, H.D. Simon and K-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs", *SIAM J. Matrix Anal. Appl.* Vol. 11, No. 3, 1990, pp. 430-452.

[21] D.E. Johnson, C.R. Aragon, L.A. McGoech and C. Schevon, "Optimization by simulated annealing: an experimental evaluation; Part I, Graph partitioning", *Operations Research*, Vol. 37, No. 6, 1989, pp. 865-892.

[22] J.W. Liu, "A graph matching algorithm by node separators", *ACM Trans. Mathematical Software* 15, No.3, 1989, 198-219.

[23] J. Dongarra and I. Duff, "A set of Level 3 Basic Linear Algebra Subprograms", *ACM Trans. Mathematical Software*, No.1, 1990, 1-17.

[24] A.H. Karp, "Programming for Parallelism," *IEEE Computer*, May 1987.

[25] H.X. Lin and H.J. Sips, "A distributed direct-solver for the DIANA finite element system", ESPRIT Genesis-P deliverable, 1991.