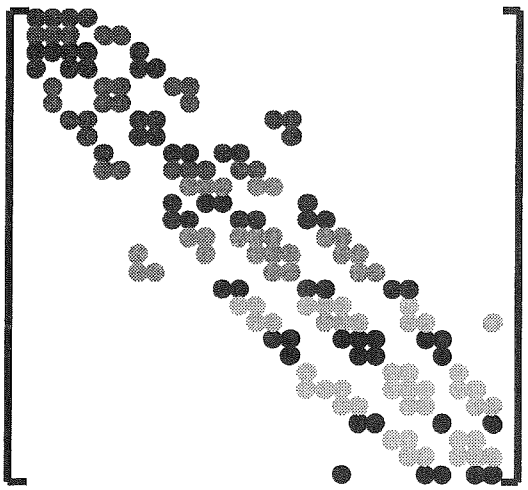
Centro Svizzero di
Calcolo Scientifico

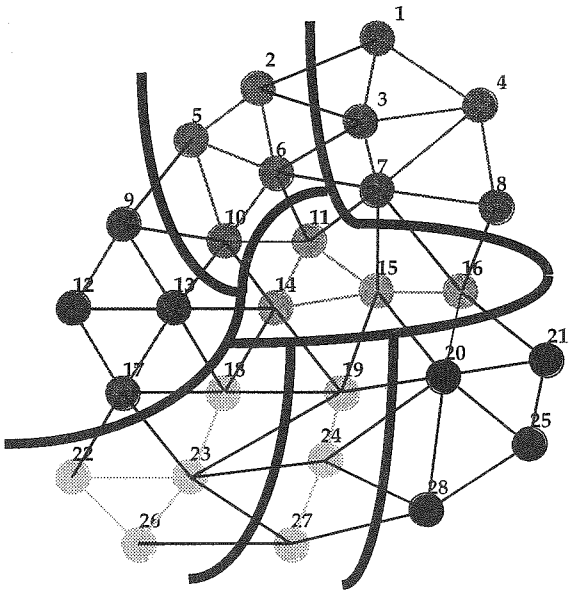
Swiss Center for
Scientific Computing

ETH

Parallel Library for Unstructured Mesh Problems (PLUMP)

O. Bröker
P. Messmer
V. R. Deshpande
W. B. Sawyer





TECHNICAL REPORT

TR-96-15

May 1996

OTHER PUBLICATIONS BY CSCS/SCSC

Annual Report:	yearly review of activities and projects
CrosSCutS (triannually):	newsletter featuring announcements relevant to our users as well as research highlights in the field of high-performance computing
Speedup Journal (biannually):	proceedings of the SPEEDUP Workshops on Vector and Parallel Computing, published on behalf of the SPEEDUP Society
User's Guide:	manual to hardware and software at CSCS/SCSC

To receive one or more of these publications, please send your full name and complete address to:

Library
CSCS/SCSC
via Cantonale
CH-6928 Manno
Switzerland

Fax: +41 (91) 610 8282

E-mail: `library@cscs.ch`

Technical Reports are also available from:
`http://www.cscs.ch/Official/Publications.html`
A list of former IPS Research Reports is available from:
`http://www.cscs.ch/Official/IPSreports.html`

Parallel Library for Unstructured Mesh Problems (PLUMP)

O. Bröker[1] **V. R. Deshpande**[2]
P. Messmer[2] **W. B. Sawyer**[2]

TR-96-15, May 1996

Abstract. The growing class of applications which solve partial differential equations (PDEs) on unstructured adaptive meshes are considered. Solution to such sparse, non-symmetric and in most cases ill-conditioned systems is often obtained using iterative methods. The programming complexity of such applications on parallel architectures is well known. The development of a Parallel Library for Unstructured Mesh Problems (PLUMP), which supports the transparent use of parallel machines for such applications, is addressed. PLUMP exploits the common denominators in such problems, provides key kernels such as the matrix-vector product and preconditioners for a wide range of iterative solvers, and supports the parallelization of this class of applications in a clean and concise manner.
The PLUMP library is implemented in C and FORTRAN77 using the Message-Passing Interface (MPI) and is available free under copyright for research purposes.

Keywords. PDE, unstructured adaptive meshes, preconditioner

This work was performed as part of the *Joint CSCS/NEC Collaboration in Parallel Processing.*

[1] German National Research Center for Computer Science (GMD),
Schloß Birlinghoven, D-53754 Sankt Augustin, Germany
[2] Swiss Center for Scientific Computing (CSCS/SCSC),
Via Cantonale, CH-6928 Manno, Switzerland

Contents

List of Figures

List of Tables

1 Introduction

There are many advantages in using unstructured meshes for field and flow solvers, including ease of automatic grid generation, ability to accurately grid complex geometries and ease of local adaption. However they require complex data structures and are less intuitive than structured grids. The growing class of applications which solve partial differential equations (PDEs) on unstructured adaptive meshes are considered. Such applications are common in various fields such as semiconductor device simulation and computational fluid dynamics (CFD). Even though unstructured grids provide the flexibility to cluster or expand the grids, the interesting problems in this class still manage to exceed the limits of today's most powerful computers. The central, most time-consuming portion of such problems is the solution of a very large system of linear equations. The matrix resulting from such a system is sparse, non-symmetric, and often ill–conditioned. Due to their large size, these systems are solved with iterative methods. Depending upon the desired accuracy of the solution, mesh refinement is done in certain regions in the physical domain, resulting in changes in the underlying data structure.

The programming complexity of such applications is well known, and results in a large development investment to port applications to parallel architectures. In view of this large effort in numerous research groups, a parallel library for unstructured mesh problems (PLUMP) is proposed to provide a transparent use of parallel machines to solve such PDE problems on unstructured grids.

To support all applications on unstructured grids would require a much too wide spectrum of functionality. By collecting common attributes and making some reasonable assumptions, PLUMP supports a large subset in a straightforward manner. The report is organised as follows - After a brief description of some related work, we introduce in Sect. 1.2, the concept of sparse matrix description through graph interpretation, which makes the design decisions of PLUMP more lucid. In Sect. 2 we consider the application requirements which the library needs to support, and enumerate the assumptions made about them. Section 3 describes the functionality of PLUMP and in Sect. 4 we present some initial results for a relatively large size problem and summarize the potential usefulness of PLUMP for other applications. The PLUMP interfaces are specified in Appendix A with a complete description of all the arguments. Appendix B contains a detailed discussion of the implementation issues of the library.

1.1 Related Work and Motivation for PLUMP

Many of today's computer simulations are described through a set of partial differential equations based on the underlying physical problem. Analytic solutions are generally not obtainable, therefore the continuous domain is approximated by a *discrete* set of interconnected points, on the assumption that the error due to this discretization is negligible. We usually refer to this set of points as a mesh or *grid*. The resulting system of discrete equations is then solved, resulting in an approximation of the analytic solution. Several discretization methods exist, e.g., the finite–difference, the finite–volume and the finite–element method.

While the grids must approximate the problem domain D, they may be structured

or unstructured, coarse or fine, as long as the set of equations can be satisfied on the given set of grid points with a desired accuracy. It is very desirable to have a grid resolution as high as possible, although using a regular grid can lead to large data sets and equations.[1] Solving such systems on computers can lead to huge storage and computation requirements. Furthermore, to obtain an answer in reasonable time, more sophisticated methods must be used, or a compromise between the accuracy of the solution and the required computing resources needs to be made.

To partially overcome the effect of the available computing resources on the desired accuracy of the solution, unstructured grids have proven very useful, especially when the physical properties are highly disparate in small portion of the domain, but nearly homogenous in a large portion. It is fruitful to work with a grid that has a high resolution in concentrated areas while maintaining a low but sufficient resolution in other regions which do not alter the final result. A good example is the investigation of shocks (see Figs. 1 and 2) created around airplane wings. Since the areas that need a high resolution are generally not known a priori, it is very useful to use *adaptive* solution methods that refine the grid based on intermediate results, while computing the solution.

Algorithms for unstructured grids are becoming increasingly popular, especially with the CFD community where the geometrical flexibility of unstructured grids enables complex geometry to be modeled easily. The resulting calculations are often huge and so there is a need to fully exploit modern parallel hardware.

Writing an individual, machine-specific parallel program is time consuming, expensive and difficult to maintain. Therefore there is a need for tools to simplify the task and generate very efficient parallel implementations. There has been various efforts by different groups to ease the parallelization task of mesh based problems. DIME from Caltech [Wil88], PARTI from ICASE [DS93] and OPlus [BCG94] from Oxford being some of the packages on distributed machines. Each one offers a range of attractive features for parallelization of unstructured mesh problems on parallel machines

The PLUMP framework has been formulated to enable parallelization of a large class of applications using unstructured adaptive grids. PLUMP exploits the common denominators in such problems, provides key kernels such as the matrix-vector product and preconditioners for a wide range of iterative solvers, and supports the parallelization of this class of applications in a clean and concise manner. It supports the manipulation of the mesh by insertion or deletion of elements and thus supports mesh adaptation. The interface to the user is through simple data structures, such as local stiffness matrices or even individual matrix entries. The user calls appropriate PLUMP procedures which create, update and manage the distributed internal representation of the sparse matrix. The key point is that sparse matrices arising from the discretization methods can be easily cast into the internal format and the iterative methods are implemented to be optimal for this storage. The present implementation is for distributed memory machines, but an implementation for shared memory machines would be straightforward.

[1] One of the applications where high resolution would be desirable is weather–forecasting. The underlying model is a set of partial differential equations (e.g., the Navier–Stokes–Equation) and the domain is the earth's atmosphere. To obtain reasonable results for a 10 day weather forecast, one needs an approximate maximum distance between discrete grid points in an order of magnitude of 1km. This leads to a number of grid points in the order of 10^7 to describe the entire earth.

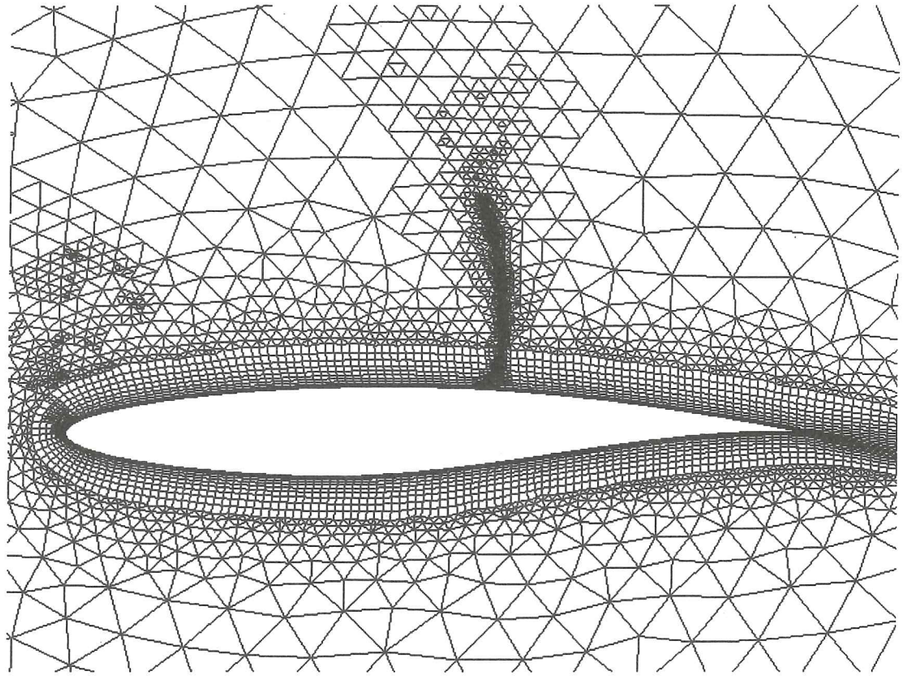


Figure 1: Mesh refinement around an aerofoil using hanging nodes.

1.2 Underlying Graph of an Unstructured Mesh Application

Irrespective of the type of PDE being solved on a grid using a given method for given boundary conditions, there is an underlying *graph* of the resulting sparse matrix which is closely related to the grid employed as shown in Fig. 3. The graph is based on a discrete set of vertices $V = \{v_1, v_2, \ldots, v_n\}$. Every vertex has a unique label $i \in \{1, \ldots, n\}$ and can always be identified through this label. In addition there are a number of interactions between vertices described by the set of directed edges, E. If these edges are weighted, then the matrix can be completely described by the graph $G = (V, E)$.[2]

In applications using grids, individual edges (i.e., matrix entries) may be too simplistic to describe the interactions between a number of vertices. For example, a finite element will define edges between all its constituent vertices. Therefore a more natural basis for such problems is to define *element graphs* to describe the interactions of larger components. A element graph consists of a subset of vertices $\mathcal{V}_k \subseteq V$ with their interactions. In the limiting case, this consist of two vertices and constitute a graph edge. There is no limitation on the number of constituent vertices of $\mathcal{V}_k$, but in practice it will be bounded implicitly by the use of a discrete operator (e.g., the type of finite elements employed). Associated with the element vertices $\mathcal{V}_k$ is a set of weighted directed edges $\mathcal{E}_k$ which describe the interactions between vertices in $\mathcal{V}_k$. The resulting element graph

[2]Weighted undirected self-edges which denote the matrix's diagonal elements must also be allowed.

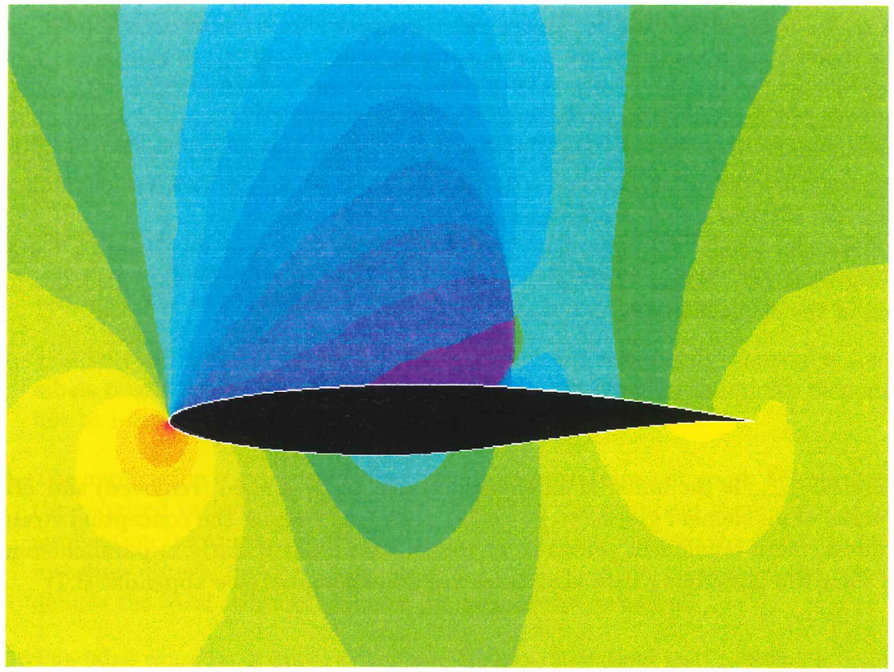


Figure 2: Mesh-based prediction of pressure field around aerofoil.

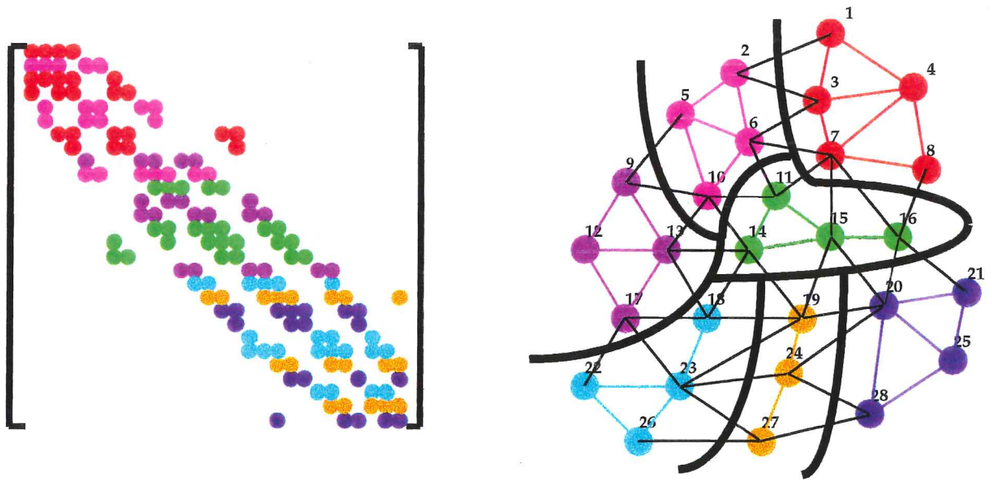


Figure 3: A matrix can be represented by its connectivity graph (at right), whose edges indicate that each that corresponding vertices mutually interact, i.e., the matrix is structurally symmetric, although the outgoing and incoming edges may be weighted different, i.e., the matrix may be non-symmetric. The graph can be partitioned (right), and the incoming edges (or matrix row entries) kept on the processing element which owns that vertex (at left).

$\mathcal{G}_k = (\mathcal{V}_k, \mathcal{E}_k)$ completely describes a element's contribution to the overall matrix.[3]

The index k enumerates the set of element graphs $EG = \cup_{k=1}^{m} \mathcal{G}_k$, where m is the total number of element graphs. Note that EG is similar but not identical to G, since EG can have multiple directed edges between two given vertices. Weights of multiple edges need to be summed to find the contribution of the single direct edge in G. It should be remembered that the graph G corresponding to EG can be determined without difficulty, and that the element graph is only an aid.[4]

The element graph EG will generally change with time, and we require a series of element-graphs $EG_1, EG_2, \ldots, EG_t, \ldots$. Mesh refinement can thus be described as a modification of the element-graph:

$$EG_{t+1} \quad = \quad (EG_t \backslash EG_{t_{\text{remove}}}) \cup EG_{t_{\text{add}}}$$

where $EG_{t_{\text{remove}}}$ is the union of all element graphs to be refined (removed) and $EG_{t_{\text{add}}}$ is the union of all element graphs to be added at time t. Despite the conceptual straightforwardness, mesh refinement proves to be difficult task, particularly in parallel environments, since the underlying data structures are quite complex (see appendix B.1).

2 Application Assumptions and Requirements

The goal of PLUMP is to offer a relatively simple library for developers of mesh-based applications to utilize the potential of parallel distributed-memory computers for the most time-consuming portion of their code: the solution of a large, sparse, non-symmetric (often structurally symmetric) system of linear equations.

For such applications, we generally assume that the inexpensive portions of the code, e.g., the insertion and deletion of elements, are running in a replicated fashion, i.e., all the data used in those portions is duplicated on all processors. This mode of operation works well if nearly all of the execution time is spent in the solver. As a direct consequence of this data replication, users can develop their code in a sequential mode and switch to parallel implementation at a later stage.

Of course, in recent years much effort has gone into parallelizing parts of finite-element applications, in particular the generation of the global stiffness matrix. Such development effort should be encouraged, and therefore PLUMP also includes several "expert" routines for applications which do not replicate all the work and data for non-solver code portions.

[3]There are several reasons why the vertices may not correspond exactly to the joints of finite elements, and the graph-elements $\mathcal{G}_k$ likewise not to the finite elements themselves. For example, sometimes "hanging" joints are used, which are involved in the element integration, but are removed before the calculation of the global matrix. The graph-elements, therefore are based only on the actual vertices of the final graph. Secondly, boundary conditions can cause joints to be removed from the final graph.

[4]The reader should be aware that other graphs can also be considered in mesh problems, e.g., the graph of connectivity between finite-elements, important for the partitioning of the problem. If the type of graph is not specified, it should be assumed that a graph of vertex connectivity is meant.

2.1 Requirements of Sequential Application

A number of assumptions need to be made about the underlying unstructured mesh application in order to decide about numerous implementation details. These assumptions resulted from an analysis of several finite-element programs, among them SESES [Kor93] and FEEL [Sug95], as well as extensive discussions with application developers.

1. All the vertices are uniquely labeled $i \in \{1 \cdots n_{\max}\}$. Likewise all the elements are uniquely labeled in the set $k \in \{1 \cdots m_{\max}\}$. An application always accesses a given vertex or element through its unique label.

2. The label to element or vertex mappings are not necessarily surjective, i.e., some labels might not refer to any vertex or element. It is the application's responsibility not to reference these undefined labels.

3. The application is expected to first specify an initial element graph EG_1 and compute the required variables on this graph. Based on the results of these calculations, the application iteratively refines the mesh, i.e., deleting some vertices and elements and inserting others, to form a new grid EG_{t+1}. At a given mesh level the application defines the local stiffness matrices for each of the elements.

4. The label must be unique (at any one time) as indicated in 1, however a given label may represent different vertex or element in different point in time. This might occur if the labels of deleted vertices or elements are reused for the insertion of new vertices or elements. On the other hand, an application which does *not* reuse labels should also be supported as memory availability permits.

5. The application may or may not keep a record of each element which it inserts, i.e., we cannot rely on the application being able to supply PLUMP with the element more than once. This implies that PLUMP cannot rely on receiving another copy of the element when, for example, the element needs to be removed from EG.

2.2 Requirements of Parallel Application

While PLUMP is designed for the transparent parallelization of a given sequential unstructured mesh problem, developers might have already given thought to their code parallelization, or even have parallelized certain parts of the code.

Any such considerations by the developers toward a parallel implementation should be exploited to the greatest extent possible. On the other hand, certain realistic requirements must be met by the parallel code to allow the use of PLUMP. We therefore make the following additional assumptions about a parallel application which calls PLUMP routines.

1. In the parallel implemention, all processing elements (PEs) must synchronously call all PLUMP routines. This is a necessary condition, as all PEs must participate in internal communication, even if one or the other PE may have no work to do.

2. The application may or may not know how the vertices and elements should be distributed to the PEs. If it does, this information should be utilized.

3. In general, the application does not know where the vertices and elements should be inserted. Therefore all PEs should have identical sets of elements and vertices upon calling the insertion routines. That is, all PEs have access to all the data, although they may choose to ignore certain data. On the other hand, if the application allows an individual PE to determine a subset of the vertices and elements, routines should be present to insert them into *EG*. Note that the latter case might occur if the element generation has been parallelized.

4. The uniqueness assumption 1 in the case of parallel implementation implies that the global labels will be mapped in some way to a PE number and to local labels on each PE. The application should only know about this mapping if it is truly necessary.

5. The application may wish to explicitly redistribute the elements and vertices. For example, if a poor load balance is determined, the application, probably assisted by repartitioning software, might wish to suggest a better distribution of the elements and vertices and may want to redistribute them explicitly. On the other hand, if the application is "naive" about the parallel implementation, PLUMP itself should decide whether redistribution is necessary.

6. If the application generates the elements in parallel, it does so in a consistent way, i.e., in a way that supports a global label space for vertices and elements. A given PE may, however, impose a local numbering scheme for given global labels for its local vertices and elements.

The assumptions 1 and 6 put some restrictions, on the use of PLUMP as some parallel implementations may work asynchronously and might not define a global label space. On the other hand, they make the implementation of PLUMP relatively easy.

3 Functionality

We consider minimal functionality as attractive for the developer, and try to support only those features which will enable a large subset of the target applications to perform well in parallel. While it is clear that additional functionality is desirable, a large class of applications could be tackled by tha functionality described in this report.

3.1 Data Initialization

Fundamental routines are called to perform the initialization of the PLUMP data structures for the vertex mapping, the global matrix and the set of elements.

3.2 Vertex Operations

Before any consideration of elements can take place, a "pool" of vertices V_I must be defined. These need to be mapped to the PEs in some way, although it is nearly impossible to choose a good mapping until the elements are defined, which gives rise to the graph G and thus the connectivity of the problem.

Vertex Insertion The application must first insert the mesh vertices V_I into the data structure. The application can either specify the PE on which a vertex should reside or PLUMP will choose a PE through a heuristic method. In addition, the "smart" parallel application may even specify the mapping of the global labels to the local labels of vertices each PE owns. Later, an incremental set of vertices V_I can be added in the same way. Note that no graph connectivity information needs to be specified at this stage. This function will create a global-to-local and global-to-PE label mapping for each vertex.

Vertex Deletion Vertex deletion frees a vertex label. The application takes responsibility for not referencing labels of deleted vertices. If a vertex is removed, the application should immediately remove all the elements which reference it. On the other hand, the application can re-insert the vertex (i.e., recycle) at any time, and thereafter insert elements which reference it. The routine to delete vertices merely resets the global-to-local and global-to-PE mappings, but does not deallocate the vertex (this would be expensive and is generally counterproductive if the vertices are being recycled).

Vertex Information The application may need to use the mapping information stored in the vertex data structure, for example, the number of vertices on a given PE, in order to perform local vector operations. In addition, the application also might need to know the global labels of certain local vector entries (i.e., determine the reverse mapping), in order, for example, to correctly describe the solution vector.

3.3 Vector Operations

Vectors are the numerical counterpart of the set of vertices and their distribution is thus determined by the vertex mapping. Typically an application will allocate and perform a wide variety of operations on many vectors. In sequential applications such vector-vector operations are commonly supported in the level-1 BLAS [LHKK79].

It is important to continue to exploit the functionality of the BLAS in the parallel code. This can be done, since vectors are still represented as an array of double precision numbers, albeit only the portion local to one PE, allowing the use of several (communication-free) BLAS routines. On the other hand, certain routines, such as scalar products and norms inherently require communication. Thus some additional functionality is needed to supplement the BLAS.

Vector Initialization Vectors often need to be set to some constant value, e.g., zeroed out. It may also be necessary to set specified elements of the vector to different values, e.g., setting to a unit vector.

Vector Norm and Scalar Product A Euclidean norm of a vector or a scalar product of two vectors requires a global summation and therefore communication. In addition, such routines need to ignore any dereferenced (deleted) vertices.

3.4 Element Operations

After a pool of vertices has been defined, the application can insert the set of elements EG_1. This set is successively altered as the mesh is refined. The application is responsible for ensuring that any vertices needed during element operations have already been inserted.

Element Insertion The application can insert new elements at any time as long as the vertices they reference exist. Either all PEs know all of the elements to be inserted (i.e., that information is replicated on every PE) or each PE has a local set to insert.

Element Deletion Elements can be removed by indicating the appropriate global element label, which is then available for reuse. As its counterpart for vertices, the corresponding routine merely resets the element's global-to-local and global-to-PE mappings, but does not deallocate the element (this would be expensive and is generally counterproductive if the elements are being recycled).

Element Refinement Element refinement could be seen as an element deletion followed by several element insertions. However, it should be supported separately since garbage collection is not needed: one of the sub-elements can replace the deleted element. Resulting load balance problems is addressed with other functionality.

3.5 Operations on the Global Matrix

Some applications may wish to access the global matrix directly. In addition to the need to insert elements directly into the global matrix, it may be necessary to insert individual matrix entries. e.g., if the matrix already exists in a triplet form of a row index, column index and a matrix entry.

3.6 Solvers

Mesh based computational applications invariably require the solution to a system $Ax = b$, which represent the discretization of the equations governing the physical processes. Since the matrix A can be extremely large, sparse and possibly non-symmetric, direct methods tend to more demanding of memory and less efficient when dealing with such problems. An appropriate way to solve the system is to use iterative solvers, e.g., linear system solvers available in the TEMPLATES [BBC+94] library. These solvers differ in their convergence properties, their robustness, the amount of memory they require, whether they need the matrix transpose (A^T), and whether A can be non-symmetric.

They require at least two things to be defined: a matrix-vector product and a preconditioner to ensure efficient convergence. We have parallelized the solvers CG, CGS, BICG,

BICGSTAB, GMRES and QMR from TEMPLATES using MPI [MPI95] and integrated them into PLUMP.

Each of these solvers consists of a data structure dependent driver, and an independent solver kernel, which communicates with the driver through a reverse communication protocol, indicating what application dependent operations, e.g., matrix-vector product or preconditioner step, it requires. The solver kernels themselves need only few alterations from the sequential version to run in parallel — they require only a parallel implementation of the scalar product and 2-norm in the case of GMRES. All other vector operations are communication free and the standard BLAS routines can be used.

Matrix-vector product A standard implementation of a sparse matrix-vector multiplication does not exhibit good data locality and uses a large amount of indirect addressing. Hence some effort was invested in the parallel implementation of the matrix-vector product to make it as efficient as possible. The matrix-vector product calculates $y \leftarrow Ax$, and requires a syntax which is compatible with the solvers. The internal data structure is updated by insertions and deletions of vertices and elements. A routine to compute the matrix-vector product first checks for the data consistency with the current set of V_l. Consistency implies that the communication pattern is up-to-date and need not be recalculated, meaning that the matrix-vector product will be as efficient as possible. Otherwise, the communication pattern is updated to make it consistent and then the product is computed.

Preconditioners Iterative methods have the disadvantage that they often do not converge (quickly) for real-life problems. This problem can be alleviated to some extent with the use of preconditioners [BBC+94], which convert the system to a related, but better-conditioned problem with improved convergence properties.

PLUMP also supplies a preconditioner PARSPAI [DGMS96] which may work well for ill–conditioned problems. Its interface has been desgined to be compatible for the parallel TEMPLATES solvers mentioned above.

3.7 Boundary Conditions

The solution of PDEs on meshes require the specification of boundary conditions to solve them. There are three possible types of boundary conditions: Dirichlet, Neumann and Robin (mixed) and their implementation in PLUMP is described below.

Dirichlet Boundary Conditions Dirichlet boundary conditions assign the function at a vertex which lies on the boundary a certain value (equal to zero if the boundary conditions are homogeneous). Conceptually, either the vertex can be removed from the system, or the corresponding row of the matrix can be "zeroed out" in the following way:

$$\hat{A} = \begin{bmatrix} * & \dots & * & * & * & \dots & * \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ * & \dots & * & * & * & \dots & * \\ 0 & \dots & 0 & 1 & 0 & \dots & 0 \\ * & \dots & * & * & * & \dots & * \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ * & \dots & * & \underbrace{*}_{k} & * & \dots & * \end{bmatrix}$$

The appropriate value b_k is then inserted to the right-hand side.

Unfortunately, since the stiffness matrices are distributed, it is not a trivial task to zero out matrix entries. Hence in the current implementation, a method of *tagging* vertices $k_1, \dots, k_m$ with boundary conditions is used such that the matrix-vector product produces the required result after filtration of these vertices. Thus,

$$\begin{bmatrix} * \\ \vdots \\ * \\ x_{k_1} \\ * \\ \vdots \\ * \\ x_{k_m} \\ * \\ \vdots \\ * \end{bmatrix} = \hat{A} \begin{bmatrix} x_1 \\ \vdots \\ x_{k_1-1} \\ x_{k_1} \\ x_{k_1-1} \\ \vdots \\ x_{k_m-1} \\ x_{k_m} \\ x_{k_m+1} \\ \vdots \\ x_n \end{bmatrix} \tag{1}$$

In particular, 1 is fulfilled for the solution vector, namely $\hat{A}x_{\text{sol}} = \hat{b}$.

The Dirichlet boundary conditions are inserted by forcing function values into the right-hand side $\hat{b}$ and tagging vertices having the boundary condition. The matrix-vector multiplication then traverses the list of Dirichlet boundary conditions to determine which entries in the multiplicand x need to be passed through and which need to be filtered out.

Neumann Boundary Conditions Neumann boundary conditions indicate the incoming or outgoing flux on the boundary, and therefore can be described with alterations to the right-hand side of the equation. The application can therefore rely on normal BLAS and PLUMP vector routines to manipulate the right-hand side as necessary.

Robin Boundary Conditions Robin or *mixed* boundary conditions impose both a Dirichlet and Neumann conditions in mesh points on the boundary. These are more difficult to handle and it is currently up to the application to separate these into Dirichlet or Neumann conditions individually (e.g., as done in SESES [Kor93]), and to use the previously discussed functionality to support them.

4 Performance Results

The performance of the PLUMP library in conjunction with the PARSPAI preconditioner was investigated on a wide variety of problems. The NEC Cenju-3, a distributed-memory machine with up to 128 nodes, each having a R4400 processor and 64MB memory was used for this purpose with the NEC-CSCS MPI library.

Very often in the calculation, a nearest-neighbor communication takes place in which a given PE needs to send very little data to only a few neighboring PEs, but keeps a large chunk of data for itself (see Fig. 4). Because of its frequent use in PLUMP, this communication kernel has been optimized using high-level MPI routines.

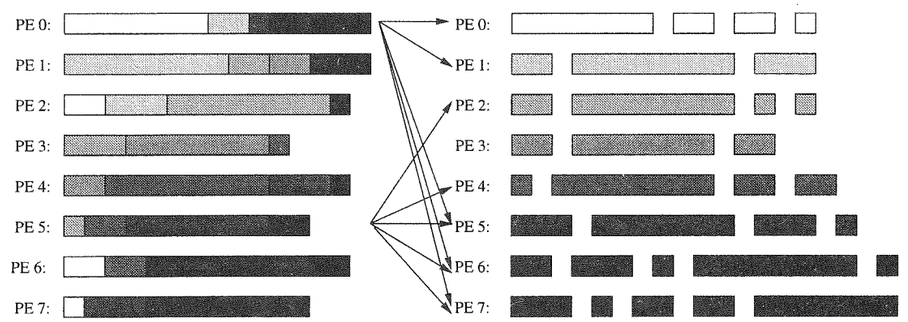


Figure 4: Commonly in applications with a high degree of data locality, a PE must send a small amount of data to a few neighboring PEs, but keeps a large chunk for itself. This operation has been optimized with non-blocking MPI communication primitives for optimum performance.

Table 1 shows the performance of the matrix-vector multiplication for a relatively large matrix ($n = 160'000$) on the Cenju-3. The efficient implementation of the matrix-vector product in indeed of utmost importance for proper scaling of the solvers.

No of Processors	8	16	32	64	128
time (ms)	224	116	67	51	35

Table 1: Cenju-3 matrix-vector product performance ($n = 160,000$ with 16 nonzeros/row).

Figure 5 shows the performance of PLUMP using the PARSPAI preconditioner and the `cgs` solver when applied to a large system of linear equations ($n = 16,384$) as resulting from PDE on a rectangular mesh. This problem could not be solved on a single processor because of memory limitations, hence the speedup is computed using 16 PEs as a base.

The scaling behavior of the solver, which depends almost entirely on the performance of the matrix-vector product and the calculation of the preconditioner, seems to be promising for tackling larger size systems. A good partitioning of the mesh, however, is important to exploit the data locality on each processor and to ensure a good compute-to-communication ratio on distributed-memory parallel machines. The performance of the preconditioner could be substantially improved by optimizing the communication, tuning the different parameters and by improvement of the basic algorithm used in PARSPAI.

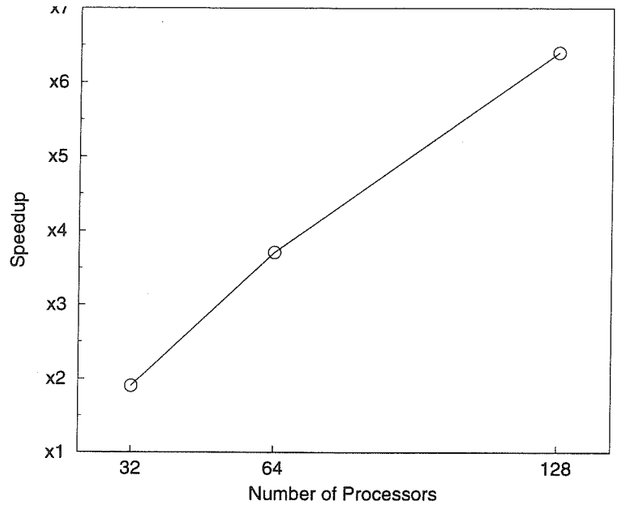


Figure 5: Speedup for solving a rectangular mesh problem on Cenju-3, $n = 16'384$

5 Possible Extensions of PLUMP

The MPI-based PLUMP is a prototype library, the design and implementation of which was based on [BLM+95]. Most of the functionality mentioned in this report has been implemented and tested. Some additional functionality could be incorporated in the library to enhance its use and is discussed in the following sections:

Eigensolvers Several applications often require that eigenvalues in a critical region and their eigenvectors be determined, e.g., λ_i and x_i in the eigenvalue equation $Ax_i = \lambda_i x_i$. Currently the most efficient way to determine these is with Arnoldi-based eigenvalue solvers, such as those available in the ARPACK library [Sor92]. Like their relatives in the TEMPLATES library, these routines require a matrix-vector product $y \leftarrow Ax$ and could be implemented in parallel based on PLUMP.

Redistribution and Remapping of Domain At some point, the application might realize that the underlying graph needs to be repartitioned in order to reduce communication, improve load balance, or both. Thus the vertices and therefore the global stiffness matrix need to be remapped onto the parallel machine. In this case, the application can supply the global-to-PE mapping for all active vertices V_l, and PLUMP will remap the vertices and then redistribute the internal matrix representation appropriately.

A future extension of PLUMP will be to interface the library to a repartitioning software package like MeTiS [KK]. Thus, the application will be able to redistribute the data transparently, i.e., without the application having to worry about which data should go where. As redistribution is expensive to perform, it will also check if the partitioning is still acceptable, as it might have been influenced only to a small extent by the addition of new elements through mesh refinement.

References

[BLM+95] I. Beg, W. Ling, A. Müller, P. Przybyszewski, R. Rühl, and W. Sawyer. PLUMP: Parallel Library for Unstructured Mesh Problems. In A. Ferreira and J. D. P. Rolim, editors, *Parallel Algorithms for Irregular Problems: State of the Art*, chapter 3, pages 45–67. Kluwer Academic Publishers, Aug. 1995. [ISBN: 0-7923-3623-2].

[BBC+94] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *TEMPLATES for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM Publications, 1994.

[BCG94] D. A. Burgess, P. I. Crumpton, and M. B. Giles. A parallel framework for unstructured grid solvers. In *Proceedings of the Second European Computational Fluid Dynamics Conference.* John Wiley and Sons, September 1994.

[DGMS96] V. Deshpande, M. J. Grote, P. Messmer, and W. Sawyer. Parallel sparse approximate inverse preconditioner. Technical Report CSCS-TR-96-14, Centro Svizzero di Calcolo Scientifico, CH-6928 Manno, Switzerland, May 1996.

[DS93] R. Das and J. Saltz. Parallelizing molecular dynamics codes using the parti software. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 187–192, March 1993.

[GH96] M. J. Grote and T. Huckle. Parallel Preconditioning with Sparse Approximate Inverses. *SIAM Journal on Scientific Computing*, 1996. To appear.

[KK] G. Karypis and V. Kumar. MeTiS: Unstructured Graph Partitioning and Sparse Matrix Ordering System. Available from `http://www.cs.umn.edu/~karypis/metis/references.html`.

[Kor93] J. Korvink. *An Implementation of the Adaptive Finite Element Method for Semiconductor Sensor Simulation.* PhD thesis, ETH-Zürich, Nov. 1993. Verlag der Fachvereine Zürich, Bericht Nr. 8.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *Transactions on Mathematical Software*, 5(3):308–323, Sept. 1979.

[MPI95] Message Passing Interface Forum (MPIF). MPI: a Message-Passing Interface standard (version 1.1). Revision of article appearing in the *International Journal of Supercomputing Applications*, 8(3/4):157–416, 1994, June 1995.

[Sor92] D. C. Sorensen. Implicit Application of Polynomial Filters in a k-step Arnoldi Method. *SIAM Journal on Matrix Analysis and Applications*, 13(1):357–385, Mar. 1992.

[Sug95] K. Sugihara. FEEL documentation. Computer Systems Research Lab., NEC Corporation, Tokyo, Japan, 1995.

[Wil88] R. D. Williams. DIME: A programming environment for unstructured triangular meshes on a distributed-memory parallel processor. In G. C. Fox, editor, *3rd Conf. on Hypercube Concurrent Computers and Applications*, volume 2, pages 1770–1787. ACM Press, Jan. 1988.

A Functional Specification

In PLUMP, a distinction is made between four fundamental types of operations: those which manipulate (insert/delete) vertices, those which work with vectors, those which manipulate elements and those which work on the global stiffness matrix — stored in a distributed ITPACK [BBC+94] storage (DIS) format. Each of these classes work on a corresponding data structure which is described thoroughly in appendix B.1.

Vertex manipulation is the first step in an application: before any elements are defined, vertices have to be specified. Subsequently, either the application can manipulate the DIS format directly, referring even to matrix entries themselves, or it can utilize the element-by-element (EBE) format which is more flexible since a record is kept of each element. The latter is needed, for example, for deletion of elements from *EG*.

A.1 Vertex Manipulation

As a first step, the application must insert vertices into a "pool" before elements can be inserted and the underlying graph treated. This can be done in several ways, depending on how much information the application can supply about the distribution of the problem.

The following routines must be called by all PEs in unison. Unless otherwise specified, all routines are called with *replicated* data. The vector handle which maintains the data mapping is discussed in appendix B.1.2.

initvertices(vertex_handle, nmax, ierr)

This routine performs all initialization necessary for the vertex mapping.

Argument	Intent	Type	Meaning
`vertex_handle`	out	see Data Formats	Handle to vertex distribution
`maxvertices`	in	integer	Maximum number of vertices
`ierr`	out	integer	Error flag

declarevertices(vertex_handle, n, ierr)

This routine is the simplest way to declare vertices, and should be used only when the total number is constant throughout and is known a priori. The heuristic used is very simple (based on the fact that neighboring vertices often have a label which is quite close). It may only be called once for a given mesh, as it initializes the mapping before inserting the vertices.

Argument	Intent	Type	Meaning
`vertex_handle`	out	see Data Formats	Handle to vertex distribution
`n`	in	integer	Total number of vertices
`ierr`	out	integer	Error flag

insertvertices(vertex_handle, n, vertexlabels, usermapping, pes, ierr)

Inserts vertices into the pool. This routine will use a heuristic to distribute the vertices to the PEs, if the application does not specify a partition of the vertices. It uses the same heuristic as in the `declarevertices` routine and will most likely lead to a non-optimal partitioning of the final mesh, implying that it will be needed to be repartitioned at some point. While this routine must be called with replicated data, it may be called multiple times as the mesh is refined. The user can also supply the global-to-PE mapping.

Argument	Intent	Type	Meaning
`vertex_handle`	inout	see Data Formats	Handle to vertex distribution
`n`	in	integer	Number of vertices to insert
`vertexlabels`	in	integer(n)	Global indices of the vertices
`usermapping`	in	logical	TRUE if user has defined `pes`
`pes`	in	integer(n)	PE ownership of corresp. vertices
`ierr`	out	integer	Error flag

mapvertices(vertex_handle, nlocal, vertexlabels, localindices, ierr)

This routine also inserts vertices into the pool and can be called multiple times. In contrast to `insertvertices` however, there is no replication of data. Each PE forces the mapping of a local number `nlocal` of globally labeled vertices to the `localindices` specified. The forced mapping means that the application can refer to individual vector entries in the local vector segment — opening numerous possibilities. However it is advisable to be used by an application which has knowledge about the underlying distribution of the data.

Argument	Intent	Type	Meaning
`vertex_handle`	inout	see Data Formats	Handle to vertex distribution
`nlocal`	in	integer	Number of local vertices to insert
`vertexlabels`	in	integer(n)	Global indices of the vertices
`localindices`	in	integer(n)	Local indices of corresp. vertices
`ierr`	out	integer	Error flag

deletevertices(vertex_handle, n, vertexlabels, ierr)

Delete the n vertices associated with the corresponding array of labels, replicated on all PEs. Note that the application is responsible for assuring that a given vertex is no longer referenced (i.e., any elements referencing the vertex have been removed). PLUMP does not actually delete the vertex (this would require too much array management), but marks it as unreferenced. The application should therefore try to "recycle" dereferenced vertices as far as possible.

Argument	Intent	Type	Meaning
`vertex_handle`	inout	see Data Formats	Handle to vertex distribution
`n`	in	integer	Number of vertices to delete
`vertexlabels`	in	integer(n)	Global indices of the vertices
`ierr`	out	integer	Error flag

dirichletvertices(vertex_handle, n, vertexlabels, ierr)

Specify the n vertices associated with the corresponding array of labels as having Dirichlet boundary conditions. These vertices will then be fully active in the application, but will be filtered in the calculation of the matrix-vector product (`disamux` and `ebeamux`) as described in Sect. 3.7.

Argument	Intent	Type	Meaning
`vertex_handle`	inout	see Data Formats	Handle to vertex distribution
`n`	in	integer	Number of dirichlet vertices
`vertexlabels`	in	integer(n)	Global indices of the vertices
`ierr`	out	integer	Error flag

numberlocalvertices(vertex_handle, result, ierr)

Returns the number of vertices in each of local PE's segment. This number also includes the dereferenced vertices, therefore caution should be exercised when making use of the returned value.

Argument	Intent	Type	Meaning
`vertex_handle`	in	see Data Formats	Handle to vertex distribution
`result`	out	integer	Number of vertices on local PE
`ierr`	out	integer	Error flag

globallabelvertices(vertex_handle, vertexlabels, ierr)

Given the vertex mapping, this routine returns the corresponding global labels which are represented by each of the local vector entries. This is required, for example, when considering the solution vector. A call to `numberlocalvertices` should preceed this call to determine how long the `vertexlabels` vector should be.

Argument	Intent	Type	Meaning
`vertex_handle`	in	see Data Formats	Handle to vertex distribution
`vertexlabels`	out	integer(*)	Global labels of local vector entries
`ierr`	out	integer	Error flag

A.2 Vector Operations

All vectors in an application are identically distributed arrays based on the vertex mapping. In general it should not be necessary for the application to manipulate these vectors directly (although this is possible if `mapvertices` has been used to define the mapping), but rather vectors can be set, read and referenced using the functionality discussed here. On the other hand, the application can call BLAS routines which do *not* require communication: first the length of the local data segment is determined with `numberlocalvertices`. Then the corresponding routine, e.g., `dscal`, `daxpy`, `dcopy` or others, can be called using the local length as the first argument. Thus the following routines are only a necessary supplement for BLAS.

The following routines must be called by all PEs in unison with replicated input arguments unless otherwise specified.

initvector(vertex_handle, constant, x, ierr)

Set all the entries of the vector x to `constant` except any dereferenced (deleted) positions, which are set to 0.

Argument	Intent	Type	Meaning
`vertex_handle`	in	see Data Formats	Handle to vertex distribution
`constant`	in	double	The values to set into x
`x`	out	double(*)	The vector x
`ierr`	out	integer	Error flag

setvector(vertex_handle, n, vertexlabels, values, x, ierr)

Set the values of x in the n positions specified by `vertexlabels` to the corresponding `values`.

Argument	Intent	Type	Meaning
`vertex_handle`	in	see Data Formats	Handle to vertex distribution
`n`	in	integer	Number of vector entries to set
`vertexlabels`	in	integer(n)	Global indices of the vertices
`values`	in	double(n)	The values to set into x
`x`	inout	double(*)	The vector x
`ierr`	out	integer	Error flag

dnrm2vector(vertex_handle, x, result, ierr)

Calculate the Euclidean norm of the vector x.

Argument	Intent	Type	Meaning
`vertex_handle`	in	see Data Formats	Handle to vertex distribution
`x`	in	double(*)	The vector x
`result`	out	double	The Euclidean norm $\sqrt{x^T x}$
`ierr`	out	integer	Error flag

ddotvector(vertex_handle, x, y, result, ierr)

Calculate the scalar product $x^T y$.

Argument	Intent	Type	Meaning
`vertex_handle`	in	see Data Formats	Handle to vertex distribution
`x`	in	double(*)	The vector x
`y`	in	double(*)	The vector y
`result`	out	double	The scalar product $x^T y$.
`ierr`	out	integer	Error flag

A.3 Distributed ITPACK Storage (DIS) Format

Within PLUMP, the data is held in an internal format which offers a high degree of transparency to the user. The distributed ITPACK storage (see appendix B) employs the basic ITPACK scheme which assumes that there are a maximum number of non-zero entries per row of the global matrix. As long as neither this maximum, nor the maximum number of rows on any one processor is exceeded, new elements are accommodated as the mesh is refined.

This distribution assures that the kernel operation for the solver $y \leftarrow Ax$ can be implemented relatively easily, and that communication can be partially overlapped with computation.

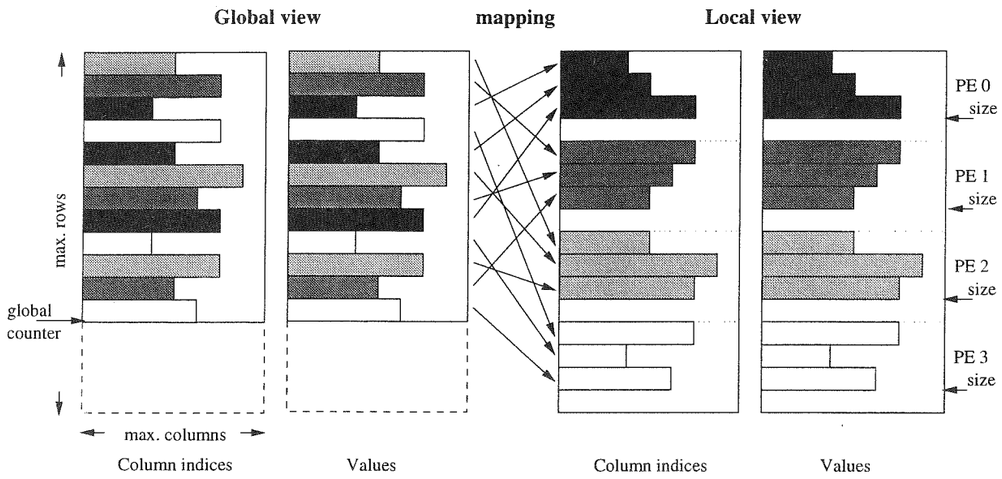


Figure 5: The distributed ITPACK storage assumes that there is a maximum number of matrix entries per row, and that the graph is adequately partitioned such that no PE fills its local two-dimensional array. For each entry in the matrix value array, there is an array containing the corresponding column indices in each row.

The following routines must be called by all PEs in unison with replicated input parameters, unless otherwise specified. The handles are discussed in appendix B.

disinit(dis_handle, maxrows, maxlocwidth, maxextwidth, ierr)

This routine performs all initialization necessary for the DIS format.

Argument	Intent	Type	Meaning
`dis_handle`	out	see Data Formats	Handle to matrix information
`maxrows`	in	integer	Max. number of rows
`maxlocwidth`	in	integer	Max. number of local neighbors
`maxextwidth`	in	integer	Max. number of external neighbors
`ierr`	out	integer	Error flag

disinsertentries(dis_handle, vertex_handle, n, rind, cind, entries, ierr)

This routine inserts individual matrix entries into the DIS format. The input is in the triplet form (row index, column index, entry value). All PEs must call this routine in unison and all the data must be replicated. The application which generates entries in a distributed fashion should call `disinsertlocalentries`.

Argument	Intent	Type	Meaning
`dis_handle`	inout	see Data Formats	Handle to matrix information
`vertex_handle`	in	see Data Formats	Handle to vertex distribution
`n`	in	integer	Number of entries to insert
`rind`	in	integer(n)	Global row index to insert
`cind`	in	integer(n)	Global column index to insert
`entries`	in	double(n)	Matrix entries corresp. to indices
`ierr`	out	integer	Error flag

disinsertlocalentries(dis_handle, vertex_handle, nlocal, rind, cind, entries, ierr)

This routine inserts individual matrix entries into the DIS format. The input format is the triplet form (row index, column index, entry value). Although all PEs must call this routine in unison, different PEs can insert different entries, i.e., the data need not be replicated. This is extremely powerful if the application can generate different parts of the global stiffness matrix independently on different PEs, but it should be used with some consideration for the vertex mapping, since it can generate enormous communication.

Argument	Intent	Type	Meaning
`dis_handle`	inout	see Data Formats	Handle to matrix information
`vertex_handle`	in	see Data Formats	Handle to vertex distribution
`nlocal`	in	integer	Number of local entries to insert
`rind`	in	integer(nlocal)	Global row index to insert
`cind`	in	integer(nlocal)	Global column index to insert
`entries`	in	double(nlocal)	Matrix entries corresp. to indices
`ierr`	out	integer	Error flag

disinsertelements(dis_handle, vertex_handle, n, size, vertices, elements, ierr)

This routine inserts elements into the graph, as described by the vertices which form its corners and its local stiffness matrix. The application is responsible for ensuring that the vertices referenced by an element have been added to the vertex pool. The application which generates elements in a distributed fashion should call `disinsertlocalelements`.

Argument	Intent	Type	Meaning
`dis_handle`	inout	see Data Formats	Handle to matrix information
`vertex_handle`	in	see Data Formats	Handle to vertex distribution
`n`	in	integer	Number of elements to insert
`size`	in	integer	Size of each element to insert
`vertices`	in	integer(size*n)	Global vertices of elements
`elements`	in	double(size^2*n)	Local stiffness matrices of elements
`ierr`	out	integer	Error flag

disinsertlocalelements(dis_handle, vertex_handle, nlocal, size, vertices, elements, ierr)

This routine inserts elements into the graph, as described by the vertices which form its corners and its local stiffness matrix. Although all PEs must call this routine in unison, different PEs can insert different elements, i.e., the data need not be replicated. The application is responsible for ensuring that the vertices referenced by an element have been added to the vertex pool.

Argument	Intent	Type	Meaning
`dis_handle`	inout	see Data Formats	Handle to matrix information
`vertex_handle`	inout	see Data Formats	Handle to vertex distribution
`nlocal`	in	integer	Number of elements to insert
`size`	in	integer	Size of each element to insert
`vertices`	in	integer(size*nlocal)	Row indices of the element
`elements`	in	double(size^2*nlocal)	Local stiffness matrices
`ierr`	out	integer	Error flag

disredistribute(dis_handle, vertex_handle, vertexlabels, usermapping, pes, ierr)

This routine redistributes the vertices specified by their global labels, and their new PE ownership. It also makes the corresponding changes to the DIS format. Currently only `usermapping` = TRUE is implemented, and the application must find the PE ownership by using a repartitioning software package externally.

Alternatively, if the subsequently described EBE format is used — which does not use the vertex mapping — it is possible to redistribute the vertices by explicit calls to the vertex manipulation routines and then to create the DIS format when it is needed with `ebetodis`.

Argument	Intent	Type	Meaning
`dis_handle`	inout	see Data Formats	Handle to matrix information
`vertex_handle`	inout	see Data Formats	Handle to vertex distribution
`vertexlabels`	in	integer(n)	Global indices of the vertices
`usermapping`	in	logical	TRUE if user has defined `pes`
`pes`	inout	integer(n)	PE ownership of corresp. vertices
`ierr`	out	integer	Error flag

disamux(dis_handle, alpha, x, beta, y, ierr)

Multiply the global matrix by a vector, scale it by α and add it to the scaled vector βy. This routine is highly optimized, overlaps communication with calculation. All PEs must call this routine in unison. The vectors x and y are distributed (see appendix B).

Argument	Intent	Type	Meaning
`dis_handle`	in	see Data Formats	Handle to matrix information
`alpha`	in	double	Scaling factor of x
`x`	in	double(*)	Incoming vector
`beta`	in	double	Scaling factor of y
`y`	inout	double(*)	Resultant vector $y \leftarrow \alpha Ax + \beta y$
`ierr`	out	integer	Error flag

disatmux(dis_handle, alpha, x, beta, y, ierr)

Multiply the transpose of the global matrix by a vector, scale it by α and add it to the scaled vector βy. This routine is highly optimized, overlaps communication with calculation, and should be used as `disamux` in solvers which require it. The vectors x and y are distributed (see appendix B).

Argument	Intent	Type	Meaning
`dis_handle`	in	see Data Formats	Handle to matrix information
`alpha`	in	double	Scaling factor of x
`x`	in	double(*)	Incoming vector
`beta`	in	double	Scaling factor of y
`y`	inout	double(*)	Resultant vector $y \leftarrow \alpha A^T x + \beta y$
`ierr`	out	integer	Error flag

disformprec(dis_handle, dis_prec_handle, spai_maxsteps, spai_epsilon, spai_beta, ierr)

Determine a PARSPAI [DGMS96] preconditioner from the global stiffness matrix. This preconditioner assumes that the matrix is *structurally* symmetric — a condition which is not assured if the matrix was inserted with `disinsertentries` or `disinsertlocalentries`. The application must specify a tolerance $0 < \epsilon < 1.0$ which determines the quality of the preconditioner — smaller the `epsilon`, the better the preconditioner. `Maxsteps` is the maximum number of steps for determining each row of the preconditioner — if the ϵ criterion is not met, PARSPAI takes the last approximation as the best estimate. The `beta` factor is used to multiply the average residual from the neighbors and choosing only those ones whose residue is less than this quantity.

Note that the `dis_prec_handle` has the same structure as the `dis_handle` and thus the preconditioner M is actually applied with a call to `disamux` to form $y \leftarrow Mx$.

Argument	Intent	Type	Meaning
`dis_handle`	in	see Data Formats	Handle to matrix information
`dis_prec_handle`	out	see Data Formats	Handle to preconditioner information
`spai_maxsteps`	in	double	Max. number of iterations per row
`spai_epsilon`	in	double	Quality factor of preconditioner
`spai_beta`	in	double	Criteria for selection of neighbors
`ierr`	out	integer	Error flag

dissolver(dis_handle, solver_method, x, y, iter, resid, prec_method, spai_maxsteps, spai_epsilon, spai_beta, info)

Solves a given linear system $Ax = y$ for x by using a given iterative solver and applying a preconditioner if required.

Argument	Intent	Type	Meaning
`dis_handle`	in	see Data Formats	Handle to matrix information
solver_method	in	integer	Selects the requested solver
x	inout	double(*)	Initial guess and resultant vector $x = A^{-1}y$
y	in	double(*)	Right hand side vector
`iter`	in	integer	Number of solver iterations
`resid`	in	double	Residue as convergence criteria
`prec_method`	in	integer	Preconditioning method
`spai_maxsteps`	in	integer	Number of refinements in SPAI
`spai_epsilon`	in	double	Quality factor of the preconditioner
`spai_beta`	in	double	Criteria for selection of neighbors
`info`	inout	integer	Status flag

To choose between the different solvers, the following values can be defined for `solver_method`:

Symbol	Value	Remark
`CG`	1	`PARSPAI` can not be applied
`CGS`	2	
`BICG`	3	
`BICGSTAB`	4	
`GMRES`	5	`info` contains number of restarts
`QMR`	6	

To choose between the different preconditioning methods, the following values are defined for `prec_method`:

Symbol	Value	Remark
`PREC_NONE`	0	No preconditioner is applied
`PREC_JACOBI`	1	Jacobi preconditioner
`PREC_SPAI`	2	PARSPAI [DGMS96] preconditioner

A.4 Element-by-element Storage (EBE) Format

The element-by-element (EBE) storage format should be thought of as an additional wrapper around the more fundamental DIS format. It is useful if the application does not keep track of the elements and needs to delete and insert elements due to mesh refinement. The elements are assigned to PEs through a very simple heuristic which does not take the vertex mapping into account. Thus a change in the vertex mapping, e.g., a remapping of the vertices will not directly affect the data in the EBE storage format.

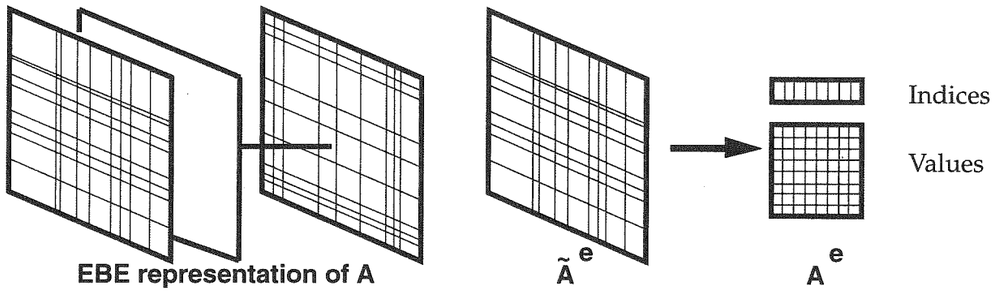


Figure 6: The EBE storage is useful for mesh manipulation at element level

Even though a matrix-vector product `ebeamux` using the EBE format is supplied, this should only be used if memory is at a premium, and performance is not a key consideration — this operation is much less efficient than its DIS counterpart. Applications which require efficient matrix-vector products should convert the elements to DIS format with `ebetodis` whenever the element graph is frozen, i.e., when EG_t is set, and then use `disamux`.

The following routines have many similarities with the vertex operations. Indeed the distribution mechanism of both is identical, and an element can be considered as a "vertex with baggage," i.e., not only do the elements require global-to-local and global-to-PE mappings, but also data structures for the set of constituent vertices and the local stiffness matrix must be kept. All routines must be called by all PEs in unison. The handles are discussed in appendix B. Unless otherwise specified all the arguments must be replicated.

ebeinit(ebe_handle, maxelements, maxsize, ierr)

This routine performs all initialization necessary for the EBE format.

Argument	Intent	Type	Meaning
`ebe_handle`	out	see Data Formats	Handle to element information
`maxelements`	in	integer	Max. number of elements
`maxsize`	in	integer	Max. vertices per element
`ierr`	out	integer	Error flag

ebeinsertelements(ebe_handle, n, size, elementlabels, vertices, elements, usermapping, pes, ierr)

Insert elements belonging to EG_t. This routine uses a heuristic to distribute the elements to the PEs, if the application does not specify a partition of the elements. The heuristic is very simple and does not depend on the vertex mapping, since that might change with time.

Argument	Intent	Type	Meaning
`ebe_handle`	inout	see Data Formats	Handle to element information
`n`	in	integer	Number of elements to insert
`size`	in	integer	Size of each element to insert
`elementlabels`	in	integer(n)	Global labels of the elements
`vertices`	in	integer(size*n)	Global vertices of elements
`elements`	in	double(size^2*n)	Local stiffness matrices
`usermapping`	in	logical	TRUE if user has defined `pes`
`pes`	inout	integer(n)	PE ownership of the elements
`ierr`	out	integer	Error flag

ebedeleteelements(ebe_handle, n, elementlabels, ierr)

Delete the n elements associated with the corresponding labels. Note that the application is responsible for first removing the elements before the corresponding vertices are deleted (dereferenced). PLUMP does not actually delete the elements (this would require too much array management), but marks them as unreferenced. The application should therefore try to "recycle" elements as far as possible, or, preferably, use the `eberefineelements` routine which avoids the management problem.

Argument	Intent	Type	Meaning
`ebe_handle`	inout	see Data Formats	Handle to element information
`n`	in	integer	Number of elements to delete
`elementlabels`	in	integer(n)	Global indices of the elements
`ierr`	out	integer	Error flag

eberefineelements(ebe_handle, n, m, size, elementlabels, vertices, elements, ierr)

This routine refines n elements into m subelements each. The same could be achieved by a set of element deletions and insertions, however it is better to use this routine, since the number of elements can only grow and no "garbage collection" is required. The application cannot specify the PE owner and there is no checking that the new vertices are correct with respect to the old element.

Argument	Intent	Type	Meaning
`ebe_handle`	inout	see Data Formats	Handle to element information
`n`	in	integer	Number of elements to refine
`m`	in	integer	Refinement elements per element
`size`	in	integer	Size of each element to insert
`elementlabels`	in	integer(n)	Global indices of elements to refine
`vertices`	in	integer(size*n*m)	Global vertices of elements
`elements`	in	double(size^2*n*m)	Local stiffness matrices
`ierr`	out	integer	Error flag

ebemapelements(ebe_handle, nlocal, size, elementlabels, vertices, elements, localindices, ierr)

This routine also inserts elements into EG_t and can be called multiple times. In contrast to `ebeinsertelements`, however, the arguments need not be replicated: each PE inserts a local number of globally labeled elements into the EBE format. This routine is therefore only to be used by an application which understands much about the underlying distribution of the data, and should be thus programmed only by expert users. This routine can also be used for refining elements, as there is no checking if existing elements are being overwritten.

Argument	Intent	Type	Meaning
`ebe_handle`	inout	see Data Formats	Handle to element information
`nlocal`	in	integer	Number of vertices to insert
`size`	in	integer	Size of each element to insert
`elementlabels`	in	integer(nlocal)	Global indices of the vertices
`vertices`	in	integer(size*nlocal)	Global vertices of elements
`elements`	in	double(size^2*nlocal)	Local stiffness matrices
`localindices`	in	integer(nlocal)	PE ownership of the resp. elements
`ierr`	out	integer	Error flag

ebeamux(ebe_handle, alpha, x, beta, y, ierr)

Multiply the global matrix by a vector, scale it by α and add it to the scaled vector βy. Note that it would be more efficient to first convert the EBE to DIS format and then perform `disamux`, since EBE format has inherent redundancy and only contains global vertex labels. On the other hand, if one cannot afford allocating the additional space, or if only very few matrix-vector products are required, it is quite reasonable to use this routine. While all PEs must call this routine in unison, the vectors x and y are distributed (see appendix B).

Argument	Intent	Type	Meaning
`ebe_handle`	in	see Data Formats	Handle to element information
`alpha`	in	double	Scaling factor of x
`x`	in	double(*)	Incoming vector
`beta`	in	double	Scaling factor of y
`y`	inout	double(*)	Resultant vector $y \leftarrow \alpha Ax + \beta y$
`ierr`	out	integer	Error flag

ebetodis(ebe_handle, vertex_handle, dis_handle, ierr)

Translate the EBE format into the DIS format. This is extremely useful if the system of linear equations is solved with an iterative method requiring many optimized matrix-vector products and the SPAI preconditioner.

Argument	Intent	Type	Meaning
`ebe_handle`	in	see Data Formats	Handle to element information
`vertex_handle`	in	see Data Formats	Handle to vertex distribution
`dis_handle`	out	see Data Formats	Handle to matrix information
`ierr`	out	integer	Error flag

A.5 Auxillary Routines

The following additional routines provide the user of PLUMP to get information about the matrix and the internal storage.

disprintmat(dis_handle, vertex_handle, ierr)

Print the global matrix with global indices as row, column and value.

Argument	Intent	Type	Meaning
`dis_handle`	in	see Data Formats	Handle to matrix information
`vertex_handle`	in	see Data Formats	Handle to vertex distribution
`ierr`	out	integer	Error flag

dismemoryinfo(dis_handle, vertex_handle, ierr)

Print information about memory usage for every PE about the number of rows, number of local and external columns, number of entries in receive and send cache, number of entries in send and receive CSR buffer in absolute and in percentage of maximum.

Argument	Intent	Type	Meaning
`dis_handle`	in	see Data Formats	Handle to matrix information
`vertex_handle`	in	see Data Formats	Handle to vertex distribution
`ierr`	out	integer	Error flag

B Implementation Issues

B.1 Internal Data Structures

The internal data structures used in PLUMP are specified in this section. Since the library is currently compiled with FORTRAN77, the data structure consists of numerous separate fields. To ensure modularity, these are declared in an *include* file (with FORTRAN and C version), which is inserted at the proper place in the application code.

With the general availability of Fortran 90, the data could be implemented as a structure which is allocated dynamically (as opposed to the static declaration in the present implementation) which further improves modularity.

B.1.1 Vertex Handle

The vertex handle consists of the global-to-local and local-to-global mappings, as well as the size of the local data segment. Due to the foreseeable large number of vertices in V, all arrays of maximum length NMAX need to be distributed. The `local2global` array is distributed exactly as any vector – thus the local PE only knows the global labels of vertices it owns. The `global2pe` and `global2local` arrays are distributed in a round-robin (or *cyclic*) fashion: PE 0 knows the mapping for global label 1, PE 1 for label 2, etc. Thus, PE α's request for a mapping of global label i is sent to the owner $mod(i, p)$ which then returns the actual coordinates *PE owner, local index*. Although this mapping can involve lot of communication, it is only done when inserting elements into DIS format or converting EBE format to DIS.

The following table shows all variables in the vertex handle. All variables are of type `integer` and are local unless specified.

Variable	Meaning
localsize(MAXPROCS)	Number of local vertices (Replicated)
global2pe(NMAX)	Global-to-PE mapping
global2local(NMAX)	Global-to-local mapping
local2global(NMAX)	Local-to-global mapping

B.1.2 Vector Handle

As mentioned in section 3.3, all vectors are distributed according to the distribution of vertices. A vector therefore consists of only a pointer to the local data segment. Its length can be determined via `numberoflocalvertices`, and the corresponding global labels found with `globallabelvertices`.

Any number of vectors can be allocated by the applications using the maximum number of vertices NMAX.

B.1.3 DIS Handle

The DIS format is an extension of ITPACK storage. In a given row, nonzero elements belonging to the PE are either local to the PE — defined by their values and column location in the two arrays `dis_loc_entries` and `dis_loc_cols` — or are non-local (external) and thus stored in the arrays `dis_ext_entries` and `dis_ext_cols`. In the latter case, the entries can be considered as graph edges which span a partition boundary.

DIS format refers only to local vertex indices. Thus, `dis_loc_cols` are the indices of vertices on the local PE. The external indices, i.e., those in `dis_ext_cols` are pointers to a lookup table `dis_perecv` or `dis_inrecv`. The array `dis_ptrecv` contains the inverse lookup mapping. The corresponding lookup table of external entries to be sent to neighboring processes is in `dis_pesend` or `dis_insend`, which can be constructed already as the elements are inserted, *if and only if* the global stiffness matrix is constructed with either `disinsertentries` or `disinsertelements` which require replicated data. These lookup tables are compressed and sorted into compressed sparse row (CSR) arrays `dis_recvia` and `dis_recvja`, or `dis_sendia` and `dis_sendja` when the actual data transfer is performed. The following table describe all variables in the DIS format. All variables are of type `integer` except for the actual matrix entries `dis_loc_entries` and `dis_ext_entries` and are local unless specified.

Variable	Meaning
dis_nrows	Current number of rows in DIS
dis_loc_ncols(MAXROW)	Number of local non-zeros per row
dis_loc_entries(MAXLOCCOL, MAXROW)	Local values
dis_loc_cols(MAXLOCCOL, MAXROW)	Local column indices
dis_ext_ncols(MAXROW)	Number of external non-zeros per row
dis_ext_entries(MAXEXTCOL, MAXROW)	External values
dis_ext_cols(MAXEXTCOL, MAXROW)	External column indices
dis_iptrecv	Size of receive lookup table
dis_perecv(MAXCACHE)	PE ownership
dis_inrecv(MAXCACHE)	Local index
dis_ptrecv(MAXCACHE)	Reverse mapping
dis_iptsend	Size of send lookup table
dis_pesend(MAXCACHE)	PE ownership
dis_insend(MAXCACHE)	Local index
dis_ptsend(MAXCACHE)	Reverse mapping
dis_recvia(MAXPROCS)	Receive count for each PE
dis_recvja(MAXCSR)	Receive list in CSR format
dis_sendia(MAXPROCS)	Send count for each PE
dis_sendja(MAXCSR)	Send list in CSR format
dis_consistent	Consistency state between updates

B.1.4 EBE Handle

The EBE handle is structured similar to the vertex handle with additional data structures to hold the constituent vertices and the local stiffness matrices. The elements are distributed in a cyclic manner which does not take the partitioning of the vertices into account, i.e., the global labels of the vertices are always used ignoring their distribution entirely. This means that operations on EBE format will be necessarily communication intensive. Ideally, EBE format should be used only when a record of inserted elements is needed and then only as a wrapper around DIS format. Whenever the EG_t is "frozen," the matrix should be translated to DIS form with a call to the `ebetodis` routine.

The following table shows all variables in the EBE handle. All variables are of type `integer` except for the actual matrix entries in `ebematrix` and are local unless specified.

Variable	Meaning
ebelocalsize(MAXPROCS)	No. of local elements (Replicated)
ebeglobal2pe(NMAX)	Global-to-PE mapping
ebeglobal2local(NMAX)	Global-to-local mapping
ebelocal2global(NMAX)	Local-to-global mapping
ebeelementsize(NMAX)	Size of given element
ebevertices(MAXBLOCK, NMAX)	Constituent vertices
ebematrix(MAXBLOCK, MAXBLOCK, NMAX)	Local stiffness matrix

B.2 DIS Matrix-Vector Product

The DIS matrix-vector multiplications `disamux` and `disatmux` are the workhorses of the solvers and are highly optimized. In the first place, a high-quality partitioning of the underlying mesh is assumed. This reduces the number of cut edges, thus minimizing the amount of data in the `dis_ext_`*`XXX`* data structure, and thereby the necessary communication. Furthermore, communication is overlapped with calculation in the following way in Ax:

1. The vector y is scaled with β.
2. Each PE determines the entries of x which are required by other PEs. These are packed into separate buffers for each PE and non-blocking MPI send requests are posted.
3. Each PE calculates the local portion $y_{tmp} = \alpha A_{local} x$, of the result.
4. Each PE receives the foreign entries of x sent in step 2.
5. Using the above data, the external portion $\alpha A_{ext} x$ is calculated, added to y_{tmp} which is then added βy.

Clearly step 3 can be performed while step 2 is pending.

Implementation of the matrix-vector product:

```
C Collect the entries which must be sent to other processes
      call LoadDoubleVector( x, xext, dis_sendia, dis_sendja )

C Post the entries to other processes in a non-blocking fashion
      call SendDoubleVector( xext, dis_sendia )

C Pre-scale y with beta
      call dscal( dis_nrows, beta, y, 1 )

C Perform local multiplication
      do irow = 1, dis_nrows
        do icol = 1, dis_loc_ncols( irow )
          xtmp( icol ) = x( dis_loc_cols( icol, irow ) )
        end do
        y( irow ) = y( irow ) + alpha * ddot( dis_loc_ncols( irow ),
     &                          xtmp, 1, dis_loc_entries(1, irow), 1 )
      end do

C Receive the needed entries from other processors and update the product
      call RecvDoubleVector( xext, dis_recvia )
      do irow = 1, dis_nrows
        do icol = 1, dis_ext_ncols( irow )
           xtmp( icol ) = xext( dis_ptrecv( dis_ext_cols(icol, irow) ) )
        end do
        y( irow ) = y( irow ) +  alpha * ddot( dis_ext_ncols( irow ),
     &                         xtmp, 1, dis_ext_entries( 1, irow ), 1 )
      end do
```


B.3 Parallel Sparse Approximate Inverse Preconditioner: PARSPAI

The PARSPAI preconditioner is a high-quality parallel preconditioner based on the SPAI [GH96] algorithm, modified for structurally symmetric matrices.

The main idea of SPAI is to approximate the inverse of the matrix A in a given norm, e.g. to reduce the residual

$$r = A * M - I.$$

If the Frobenius Norm is chosen as the criteria, then the above problem splits into n independent least squares problems, which can fully be solved in parallel:

$$\min \|A * M - I\|_F^2 = \sum_{k=1}^{n} \min_k \|A * m_k - e_k\|_2^2 = \sum_{k=1}^{n} \min_k \|r_k\|_2^2 .$$

The main problem consists in determining a good sparsity pattern for m_k. The PARSPAI algorithm starts with a diagonal pattern as initial sparsity and computes an initial residual r_k. It then tries to add those elements to m_k, which help to reduce the initial residual by maximum amount. This is done by solving a 1-dimensional minimization problem. With this new sparsity pattern, the least squares problem is solved again to decrease the residual still further. This loop is repeated until the residual–norm satisfies a given criterion $\|r_k\|_2 \leq \epsilon$, or the number of refinements exceed a given limit.

As this process has to be done for every column m_k of M and as the determination of every column is independent, the algorithm scales very well, as long as the additional communication overhead does not exceed the computational time.

Implementation of PARSPAI and performance results on various problems are discussed in [DGMS96].

C Example

```
cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
C
C This example program shows the use of PLUMP calls to insert vertices
C and elements in a 2-D rectangular mesh and to insert element stiffness
C matrices in the DIS data structure used by PLUMP. It then solves the
C system of linear equations Ax=B by using parallel iterative solver CGS.
C The system is preconditioned using the PARSPAI preconditioner.
C
C ---------------------------------------------------------------------
      program grid
C ---------------------------------------------------------------------
      implicit none
C
C Include the MPI and plump related include files
C
#include "mpi_context.incl"
#include "plump.incl"
#include "dis_declaration.incl"
#include "vertex_declaration.incl"

C-Define the mesh size & no of processors in X direction
#define  PRB_XSIZE        100
#define  PRB_YSIZE        100
#define  PE_NUM_X         4
#define  PI               3.141592654

      integer total_no_of_vertices, err

C-------------------------------------------------------------
      call mpi_init(err)
      call mpicontext
      if(myid.eq.0)print *, "Benchmarking BIG problems"
      total_no_of_vertices = PRB_XSIZE * PRB_YSIZE


C Initialize the PLUMP vertex data structure and get back a handle

      call initvertices(
#include "vertex_handle.incl"
     &    total_no_of_vertices, err)

C Call a routine to insert vertices in the pool, ie. to map them
C according to the partition of the mesh (userdefined or PLUMP defined).

       call setvertices(
#include "vertex_handle.incl"
     &    PRB_XSIZE, PRB_YSIZE, PE_NUM_X,err)

C One can also call the following routine to declare the vertices.
c      print *, "declarevertices"
c      call declarevertices(
c#include "vertex_handle.incl"
c     &                  total_no_of_vertices,
c     &                  err
c     &                        )

C Initialize the DIS structure
      call disinit(
#include "dis_handle.incl"
     &               total_no_of_vertices, MAXLOCCOL, MAXEXTCOL,
     &               err
     &                  )
```



```
C
C Call a routine to generate the element stiffness matrices and solve the
C system of linear equations.

      call test(
#include "dis_handle.incl"
#include "vertex_handle.incl"
     &          err )

      end

C ----end-of-grid---------------------------------------------------




C ---------------------------------------------------------------------
      subroutine test(
#include "dis_handle.incl"
#include "vertex_handle.incl"
     &         err )
C ---------------------------------------------------------------------
      implicit none
C ---------------------------------------------------------------------

#include "mpi_context.incl"
#include "plump.incl"
#include "dis_declaration.incl"
#include "vertex_declaration.incl"
#include "solver_meth.h"

#define  ELEMENT_SIZE      4
#define  VECTOR_SIZE       (PRB_XSIZE*PRB_YSIZE)
#define  dis_max_nrows     (PRB_XSIZE*PRB_YSIZE)
#define  PI                 3.141592654

C ---------------------------------------------------------------------
      double precision x( VECTOR_SIZE ), y( VECTOR_SIZE )
      double precision dis_checksum, sum, gsum
      integer irow,isrow, iscol
      integer prb_xsize, prb_ysize, ix, iy, dx, dy
      parameter(prb_xsize = PRB_XSIZE, prb_ysize = PRB_YSIZE )
      double precision slide(  ELEMENT_SIZE, ELEMENT_SIZE ), cnt

      integer          s_row( ELEMENT_SIZE )
      integer          slide_row( ELEMENT_SIZE, ELEMENT_SIZE )
      integer          slide_col( ELEMENT_SIZE, ELEMENT_SIZE )
      integer          solver_method, info, prec_method, SPAI_maxsteps
      double precision resid, SPAI_epsilon, SPAI_beta
      integer          k, iter
      integer          err

      integer iseed(4)
      integer entr
      integer row_buf( MAXLEN ), col_buf( MAXLEN )
      double precision entry_buf( MAXLEN )

      double precision parddot
      external parddot

C ---------------------------------------------------------------------
```

```
C  Generate a matrix problem with rectangular grid

#define  nodeind(ix, iy)   (ix + (iy-1) * prb_xsize)

      cnt = 1.0d0
      entr = 0

      do 510 iy = 1, prb_ysize-1
         do 520 ix = 1, prb_xsize-1

C  Generate a slide, ie. element stiffness matrix
          isrow = 1
          do dx = 0, 1
            do dy = 0, 1
              s_row( isrow ) = nodeind( ix+dx, iy+dy )
              isrow = isrow + 1
            end do
          end do

c    *** Real values of the slide ***
          do isrow = 1, ELEMENT_SIZE
             do iscol = 1, ELEMENT_SIZE
               slide_row( isrow, iscol ) = s_row( isrow )
               slide_col( iscol, isrow ) = s_row( isrow )

ccc now some data has to be inserted in the structure, e.g. the edges have
ccc to be weighted. On way is to use a kind of Hesseberg matrix

               slide( isrow, iscol ) = 1.0d0/(cnt+isrow+iscol)

             end do
          end do
          cnt=cnt+1.0d0

c
c Insert the buffer of element matrices (if they are full) into the PLUMP DIS
c format
c
       if(entr + ELEMENT_SIZE * ELEMENT_SIZE .ge. MAXLEN) then
        call disinsertentries(
#include "dis_handle.incl"
#include "vertex_handle.incl"
     &    entr, row_buf, col_buf, entry_buf, err )
         entr = 0
       end if

c
c Pack the new element matrices into the buffers so that we can insert them
c  simultaneously in the DIS format
c
       do isrow = 1, ELEMENT_SIZE * ELEMENT_SIZE
          entr = entr + 1
          row_buf(entr)   = slide_row( isrow )
          col_buf(entr)   = slide_col( isrow )
          entry_buf(entr) = slide( isrow )
       end do

 520    end do
 510  end do
```



```
c
c Insert the buffer into the DIS format in PLUMP
c
        call disinsertentries(
#include "dis_handle.incl"
#include "vertex_handle.incl"
     &    entr, row_buf, col_buf, entry_buf, err)


C Initialize the vectors
      do 730 irow = 1, dis_max_nrows
       x(irow) = 1.d0
       y(irow) = 0.d0
 730  end do


C
C Do Matrix-vector multiplication with the matrix in DIS format to check
C the integrity of the matrix.
C
      call disamux (
#include "dis_handle.incl"
     &    1.0d0, x, 0.0d0, y, err)


C Check the 2-norm of the vector generated by matrix-vector product.

      call dnrm2vector(
#include "vertex_handle.incl"
     &     y, dis_checksum, err)

      dis_checksum = parddot(VECTOR_SIZE, y, 1, y, 1 )

C Generate the right-hand side vector y=A*x

      do 950 k = 1, VECTOR_SIZE
          x(k) = 1.0d0
          y(k) = 0.0d0
 950  continue

      call disamux (
#include "dis_handle.incl"
     &    1.0d0, x, 0.0d0, y, err)

C Give the initial guess

      do 960 k = 1, VECTOR_SIZE
          x(k) = k
 960  continue


C Print the input matrix in matlab format if required

      call disprintmat(
#include "dis_handle.incl"
#include "vertex_handle.incl"
     &    err)

C Get the memory info about the Internal DIS data structure
       call dismemoryinfo(
#include "dis_handle.incl"
#include "vertex_handle.incl"
     &    err)
```

```
C Specify the PARSPAI preconditioner parameters and the solver to be used.
      solver_method = CGS
      iter = 8000
      info = 100
      resid = 1.0d-08
      prec_method = PREC_SPAI
      SPAI_epsilon = 0.1
      SPAI_maxsteps = 10
      SPAI_beta =1.0d0 + 1.0d-10

c Solve the system

      call dissolver(
#include "dis_handle.incl"
     $    solver_method, x, y, iter, resid, prec_method,
     $    SPAI_maxsteps, SPAI_epsilon, SPAI_beta, info)

c-----------End of subroutine test ----------------------------------

      subroutine setvertices(
#include "vertex_handle.incl"
     &                       meshx, meshy, nx, err)

c Determine a partitioning of the mesh. The following divisions must
c have no remainder: nprocs/nx, meshy/ny, meshx/nx

#include "mpi_context.incl"
#include "vertex_declaration.incl"
      integer meshx, meshy, nx, err
      integer ny

      integer vertexlabels( MAXLEN ), pes( MAXLEN )
      integer iproc, ype, ydim, xdim, i, pestart,  entr, j
      logical usermapping

      usermapping = .true.
      entr = 1
      ny = nprocs / nx
      ydim = meshy / ny
      xdim = meshx / nx

       do iproc = 0, nprocs - 1
          ype = (iproc / nx)
          pestart = ype * meshx * ydim + mod(iproc, nx)*xdim + 1
          do i = 0, ydim - 1
             do j = 0, xdim - 1
               vertexlabels(entr) = pestart + i * meshx + j
               pes(entr) = iproc
               entr = entr + 1
               if(entr .gt. MAXLEN) then
                   call insertvertices(
#include "vertex_handle.incl"
     &    entr - 1, vertexlabels, usermapping, pes, err)
                   entr = 1
               end if
             end do
          end do
       end do

       call insertvertices(
#include "vertex_handle.incl"
     &    entr - 1, vertexlabels, usermapping, pes, err)
       return
       end
c-----------End of subroutine setvertices-------------------------------
```


1995

TR-95-03 C. Clémençon, K. Decker, V. Deshpande, A. Endo, J. Fritscher, N. Masuda, A. Müller, R. Rühl, W. Sawyer, B. J. N. Wylie, and F. Zimmermann: Tool-Supported Development of Parallel Application Kernels. (April 1995)

TR-95-04 Y. Seo, T. Kamachi, K. Suehiro, M. Tamura, A. Müller, and R. Rühl: Kemari: a Portable HPF System for Distributed Memory Parallel Machines. (June 1995)

TR-95-05 A. Endo and B. J. N. Wylie: Annai/PMA Instrumentation Intrusion Management of Parallel Program Profiling. (November 1995)

TR-95-06 P. Ackermann and U. Meyer: Prototypes for Audio and Video Processing in a Scientific Visualization Environment based on the MET++ Multimedia Application Framework. (June 1995)

TR-95-07 M. Guggisberg, I. Pontiggia and U. Meyer: Parallel Fractal Image Compression Using Iterated Function Systems. (May 1995)

1996

TR-96-01 W. P. Petersen: A General Implicit Splitting for Stabilizing Numerical Simulations of Langevin Equations. (February 1996)

TR-96-02 C. Clémençon, K. M. Decker, V. R. Deshpande, A. Endo, J. Fritscher, P. A. R. Lorenzo, N. Masuda, A. Müller, R. Rühl, W. Sawyer, B. J. N. Wylie, F. Zimmermann: Tools-supported HPF and MPI Parallelization of the NAS Parallel Benchmarks. (March 1996)

TR-96-03 B. J. N. Wylie and A. Endo: Annai/PMA Multi-level Hierarchical Parallel Program Performance Engineering. (April 1996)

TR-96-04 C. Clémençon, A. Endo, J. Fritscher, A. Müller, and B. J. N. Wylie: Annai Scalable Run-time Support for Interactive Debugging and Performance Analysis of Large-scale Parallel Programs. (April 1996)

TR-96-05 M. Roth: A Visualization System for Turbomachinery Flow. (April 1996)

TR-96-06 W. P. Petersen: Some evaluations of Random Number Generators in real*8 Format. (April 1996)

TR-96-07 Tetsuya Takaishi: Heavy quark potential and effective actions on blocked configurations. (April 1996)

TR-96-08 Norio Masuda and Frank Zimmermann: PRNGlib: A Parallel Random Number Generator Library. (May 1996)

TR-96-09 M. C. Hohenadel and P. Pagny: X-OpenWave User's Manual. (May 1996)

TR-96-10 Edgar A. Gerteisen: Automatized Generation of Block-Structured Meshes for a Parametric Geometry. (May 1996)

TR-96-11 V. Deshpande, W. Sawyer, and D. W. Walker: An MPI Implementation of the BLACS. (May 1996)

TR-96-12 R. Sampath, J. Fritscher, and B. J. N. Wylie: Port of the Annai tool environment to workstation clusters. (May 1996)

TR-96-13 P. A. R. Lorenzo, A. Müller, Y. Murakami, and B. J. N. Wylie: High Performance Fortran interfacing to ScaLAPACK. (May 1996)

TR-96-14 Vaibhav Deshpande, Marcus J. Grote, Peter Messmer, and William Sawyer: Parallel Sparse Approximate Inverse Preconditioner. (May 1996)

CSCS/SCSC — Via Cantonale — CH-6928 Manno — Switzerland
Tel: +41 (91) 610 8211 — Fax: +41 (91) 610 8282

CSCS/SCSC — ETH Zentrum, RZ — CH-8092 Zürich — Switzerland
Tel: +41 (1) 632 5574 — Fax: +41 (1) 632 1104

CSCS/SCSC WWW Server: http://www.cscs.ch/