Tool-box for parallel adaptive computations of 3-D convection-diffusion problems using domain decomposition

S.Z. Tomov *

June 1, 2000

Abstract

In this report we describe the tools that we have used, developed, and implemented in a computer system for simulation of flows in porous media. Our goal was to create a simulator that uses various tools and that is based on discretization by finite elements and finite volumes and uses efficient preconditioning iterative methods for the resulting large sparse system. Also important features are error control, adaptive grid refinement and parallel implementation on multiprocessor computer systems utilizing the concept of domain decomposition. The tools include *(1) 3-D mesh generator (NETGEN)*, *(2) partitioning and load balancing software (METIS)*, *(3) local error control and refinement procedures*, *(4) preconditioning methods based on domain decomposition and multigrid/multilevel*, and *(5) MPI and the OpenMP standards for massively parallel computations* .

1 Introduction

Discretization of fluid flows in 3-D gives rise to extremely large scale computation. Since the finite element and finite volume discretizations lead to sparse systems it is natural to consider iterative methods for their solution. The storage to assemble the global matrix (in sparse format) is the same in both direct and iterative solution methods, but the direct methods may produce an arbitrary amount of fill-ins that for 3-D problems might be prohibitively high. Also, fundamental physical limitations on the computer processing speed may require the exploitation of parallelism. Since matrix-vector operations can be efficiently parallelized all iterative solvers can highly benefit the parallelism, while the direct methods need sophisticated techniques to extract limited parallelism. One disadvantage of the iterative methods is that the rate of convergence is problem dependent

*This work has been partially supported by the US Environmental Protection Agency under Grant R 825207 and may be unacceptably slow. This stresses on one of the main aspects of the present work, namely acceleration of the iterative methods by preconditioning techniques.

The above brief motivation for the direction of our study partially explains the tools that we may need. First, since we discretize partial differential equations by finite element and/or finite volume methods, we need a tool for generating a finite element splitting of a given 3-D domain (described in Section 2). Our computational strategy relies on the use of a mesh generator for the coarse mesh that describes the problem and satisfies some regularity requirements. Then, based on an a posteriori error analysis, the initial grid is refined via generation of a sequence of nested meshes until a "good" one that resolves the solution adequately is found. This is done by another tool that we have developed. Its description is given in Section 5.

In our brief motivation we mentioned that for the success of an efficient computational technology the employment of parallel computations may be essential. The tool addressing this issue is given in Section 6. We discuss our strategy and give some useful information for implementing parallel finite element/volume software using the Message Passing Interface library (*MPI*) and the *OpenMP* standard.

When iterative methods are parallelized on a multiprocessor system the data distribution and the communication scheme are of greatest importance for an efficient execution. Both the data distribution and the communication scheme are usually determined before the execution of the solver by preprocessing. There are many ways to do this. It seems that the most popular and efficient is the use of *Domain Decomposition* techniques. Here by *Domain Decomposition* we first mean sub-domain splitting techniques (a tool for sub-domain partitioning is given in Section 3). Since the sub-domains are going to be mapped on different processors the goal (concerning the data distribution) is to split the domain so that a good load balancing among the processors and (concerning the communication) a small ratio of communication over computation is achieved. Second, *Domain Decomposition* techniques addresses the question of preconditioning. These are undoubtedly one of the best known and promising methods, which also take advantage of the parallelism. The tool dealing with this type of preconditioning is given in Section 4. The multilevel structure obtained in the process of consecutive local refinements is exploited by implementing multigrid preconditioners for the sub-domain solvers, which is explained in the *Domain decomposition* section.

Having defined the main tools, below we summarize the overall computational strategy. We use a 3-D mesh generator (*NETGEN*) to generate a good coarse mesh. Then the considered problem is solved sequentially on the coarse mesh. The solution is used to compute a posteriori error estimates, which are used as weights in an element based splitting of the coarse mesh into sub-domains (using *METIS*). Such splitting insures that the local refinements that follow will produce computational mesh with number of tetrahedrons balanced over the sub-domains. Every sub-domain is "mapped" to a processor. Then, based on a posteriori error analysis, each processor refines consecutively its region independently. After every step of independent refinement there is communication between the processors in order the mesh on that level to be made globally conforming.

Domain decomposition technique is used to solve the global problem. This leads to independent computations on the sub-domains (performed by the corresponding processors) and involves transfering interface data when necessary. For the local sub-domain solvers we have implemented multigrid/multilevel preconditioners.

2 Mesh generation in 3-D (NETGEN)

Finding a "good" computational mesh is one of the key elements in the development of any efficient computational methodology based on finite element or finite volume method. Both methods require partitioning the domain of interest into a set of elements, which have to possess certain regularity properties. As we mentioned in the introduction, our computational strategy (regarding mesh generation) is to use automatic mesh generation to get a good coarse mesh (satisfying some regularity requirements) and a posteriori error analysis to lead the process of obtaining the final mesh (see Section 5).

Mesh generator, which has good capabilities for generating coarse meshes and which we use, is *NETGEN*. It's a 3-D stand-alone mesh generator based on advancing front method. The splitting is into tetrahedra. *NETGEN* is developed by *Joachim Schöberl*, Johannes Kepler University, Linz, Austria. More information (than the presented below) about this mesh generator can be found on *Joachim Schöberl*'s homepage [1]. The software is free for non-commercial applications and is available for *Unix/Linux* and *Windows 98/NT* platforms.

The input is 3-D domain described by boolean operations (*or*, *and*, *not*) on primitives, such as planes, cylinders, spheres, cones and tubes. The primitives are used in the following formats:

- `plane(` p_x `,` p_y `,` p_z `;` n_x `,` n_y `,` n_z `)` – half-plane given by a point p on the plane and the outside normal vector n;
- `cylinder(` a_x `,` a_y `,` a_z `;` b_x `,` b_y `,` b_z `;` r `)` – cylinder of infinite length, given by two points a and b on the axis and radius r;
- `sphere(` c_x `,` c_y `,` c_z `;` r `)` – sphere given by center c and radius r;
- `cone(` a_x `,` a_y `,` a_z `;` r_a `;` b_x `,` b_y `,` b_z `;` r_b `)` – cone given by the two points a and b on the axis and their two corresponding radii;
- `tube(` $p_{1,x}$ `,` $p_{1,y}$ `,` $p_{1,z}$ `, ... ,` $p_{m,x}$ `,` $p_{m,y}$ `,` $p_{m,z}$ `;` r `)` – tube given by an algebraic rational spline curve of order 2 and a global radius r. The spline curve consisting of n pieces is given by $2n + 1$ points, such that patch i is determined by points $2i - 1$, $2i$ and $2i + 1$.

[1]`http://www.sfb013.uni-linz.ac.at/~joachim/`

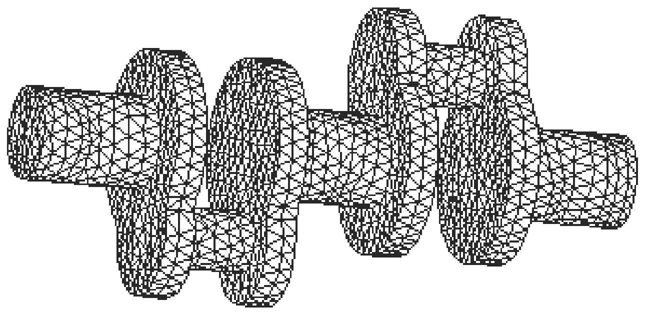

Figure 1: Crankshaft mesh. The domain is split into $5,830$ elements, with $4,176$ surface elements and $2,140$ vertices.

Using the mentioned above boolean operations one may combine the primitives to form solids, which may be used to form more complicated ones. The solid called `all` is the final object. A simple example for generating a *L-shaped domain* follows.

```
algebraic3d
solid c1 =   plane (  -1,   -1,  -1;  0,  0, -1)
         and plane (  -1,   -1,  -1;  0, -1,  0)
         and plane (  -1,   -1,  -1; -1,  0,  0)
         and plane (   1,    1,   1;  0,  0,  1)
         and plane (   1,    1,   1;  0,  1,  0)
         and plane (   1,    1,   1;  1,  0,  0);
solid c2 =   plane (   0, -1.1,   0;  0,  0, -1)
         and plane (   0, -1.1,   0;  0, -1,  0)
         and plane (   0, -1.1,   0; -1,  0,  0)
         and plane ( 1.1,    0, 1.1;  0,  0,  1)
         and plane ( 1.1,    0, 1.1;  0,  1,  0)
         and plane ( 1.1,    0, 1.1;  1,  0,  0);
solid all = c1 and not c2;
```

Note that the cube represented by solid `c2` (the one that is subtracted from `c1` in order to get the *L-shaped domain*) is "bigger" than "necessary". This is done in order for *NETGEN* to be able to compute the necessary intersection (if for solid $c2$ points $(0,-1,0)$ and $(1,0,1)$ are used instead of correspondingly $(0,-1.1,0)$ and $(1.1,0,1.1)$ then *NETGEN* wouldn't be able to split the domain). Another interesting point in generating the input file, with which one has to be careful, may be demonstrated with the following example that comes from a fluid flow domain generation. We have generated solids *domain* and *layer*. Then the difference between

```
solid all1 = layer;
solid all2 = domain and not layer;

solid all1 = domain and layer;
solid all2 = domain and not layer;
```

is that in the first case the boundary between the *layer* and and the rest of the *domain* is non-conforming. For both cases the tetrahedrons in the *layer* have attribute 1 and 2 for the rest of the domain.

An example showing *NETGEN*'s capabilities of dealing with more complicated domains is given on Figure 1.

3 Sub-domain partitioning (METIS)

Domain decomposition methods are based on a technique called *divide-and-conquer*. The main idea in this concept is that the global problem is split into sub-problems, which are solved concurrently and the local solutions are merged or combined in order to get the global one. In *Domain decomposition* (given in the next section) the above idea translates into finding a splitting $\{\mathcal{T}_{i,h}\}_{i=1}^{N}$ of the global mesh $\mathcal{T}_h$, mapping every $\mathcal{T}_{i,h}$ to a processor, doing independent computations on each $\mathcal{T}_{i,h}$ and transfering data when necessary. Crucial for the efficient parallel execution of software based on this technique is obviously the quality of the splitting $\{\mathcal{T}_{i,h}\}_{i=1}^{N}$.

In order to have load balance over the processors the number of tetrahedrons in each $\mathcal{T}_{i,h}$ should be almost equal. On the other hand, in order to reduce the communication the number of nodes on the boundary between the sub-domains should be minimal. Although this problem is *NP-complete* many relatively simple and effective heuristic methods have been devised (see the survey [7]).

In our tool-box we have used *METIS*, a software package for partitioning large irregular graphs/meshes and computing fill reducing orderings of sparse matrices. This software has been developed by *George Karypis* and *Vipin Kumar* from the University of *Minnesota*. The algorithms in *METIS* are based on multilevel graph partitioning, where the graph is consecutively coarsen, the coarse graph partitioned and then the computed partitions projected into the fine graph.

We also have used the graph partitioning part of *METIS*, where the vertices of the graph are the finite elements. In our case the vertices represent the set of tetrahedrons and the graph cliks are the common feces. *METIS* provides two programs to partition such graphs into k equal parts: *pmetis* and *kmetis*. The first one, *pmetis* is based on multilevel recursive bisection algorithm (see [10]) and is preferable for partitioning graphs into a small number N of sub-domains. The second one, *kmetis*, is based on k-way partitioning (see [9]) and is prefered when the graph has to be split into more than eight parts. Computationally our observations are that both programs produce well balanced meshes with *pmetis* being better for small values of N and *kmetis* being slightly better for large values of N. Unfortunately, often *kmetis* produces disconnected domains, especially for large N. In general, *pmetis* produces "smoother" and "better connected" domains.

Figure 2 shows the domain from Figure 1 split into 4 by *pmetis*. The domain is split at places where one intuitively would do it in order to have balance of the elements in the sub-domains and minimum interface between the sub-domains.

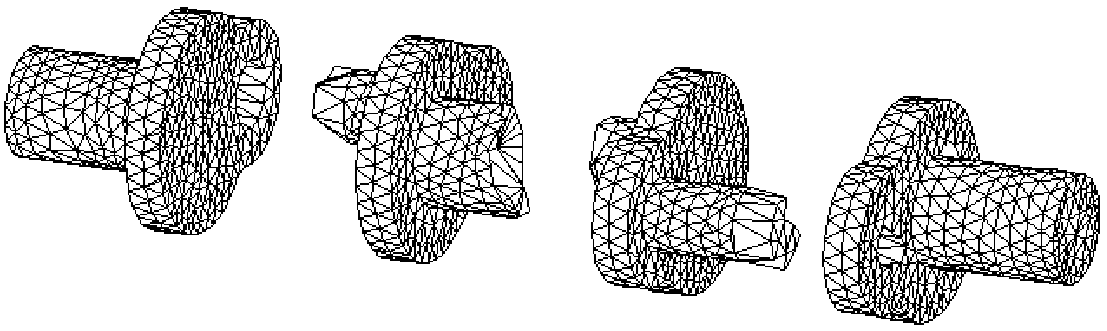

Figure 2: Crankshaft mesh split in 4 by *pmetis*. The partitions starting from left have correspondingly 1458, 1457 , 1458 and 1457 tetrahedrons.

Program *pmetis* is invoked (similarly *kmetis*) by providing file with the graph of the mesh and number of partitions that is desired. The format is

```
pmetis  GraphFile  Nparts
```

`GraphFile` may represent weighted or unweighted graph. The format for unweighted graph is

```
< # of vertices >  < # of edges >
< list of vertices with which node 1 is connected >
< list of vertices with which node 2 is connected >
 .  .  .
```

where `vertices` in our case correspond to tetrahedrons and `edges` to interior faces.

As we mentioned in the introduction part, we would like to be able to give weights on the tetrahedrons. These weights will come from a posteriori error analysis. Graph partitioner that allows weights than will split the domain such that the error in the subregions to be approximately the same. This insures that the local refinements that follow will produce computational mesh with number of tetrahedrons balanced over the sub-domains.

Weights on the tetrahedrons in *METIS* can be specified either on the faces (i.e. graph's edges) or on both tetrahedrons and faces (i.e. graph's vertices and edges). For the first case after every vertex a weight follows and the first line is

```
<# of vertices> <# of edges> 1
```

For the second a number at the beginning of every row is added giving the weight for the element. The first row is also changed and the format is as follows.

```
<# of vertices> <# of edges> 11
<node 1 weight> <list of '<vertex> <weight>' for vertices connected to 1>
  .   .   .
```

Since often only the mesh file is available *METIS* also provides two auxiliary programs, *mesh2nodal* and *mesh2dual*, for converting a mesh into the described above graph format.These programs convert triangular, tetrahedral or hexahedral (bricks) meshes into graphs suitable correspondingly for vertex based partitioning and element based partitioning. The mesh file format is

```
<# of elements> <type>
<list of vertices for element 1>
  .   .   .    ,
```

where type is 1, 2, 3 or 4 for correspondingly triangles, tetrahedra, hexahedra (bricks) or quadrilaterals. The node indexing starts from 1.

The output, a partition file of a graph with n vertices, consists of n lines with a single number per line. The i-th line of the file contains the partition number that the i-th vertex belongs to. Partition numbers start from 0.

The various programs provided in *METIS* can also be directly accessed from a *C/C++* or *Fortran* program by using the stand-alone library *METISlib*. The interface to *METISlib* routines and more information about *METIS* can be found on *METIS*'s homepage [2]. The software is free, written entirely in *ANSI C*, and is portable on most *Unix* systems.

4 Domain decomposition

In this section we describe the data structures for the non-overlapping domain decomposition methods. We start by briefly describing the Schur complement method and then we explain the implemented data structures and how they utilize the main steps used in the algorithm. Test results are given in Section 6, where the proposed data structures were tested on a parallel implementation of the conjugate gradient method.

The matrix point of view of the Schur complement method is given as follows. The discrete problem can be written in matrix form as $Ax = f$. We order the finite element/volume unknowns into two groups: the unknowns inside the sub-domains, denoted by x_D and the unknowns on the sub-domain interface denoted by x_Γ. Now we rewrite the original problem in block form

$$\begin{pmatrix} A_D & A_{D\Gamma} \\ A_{\Gamma D} & A_\Gamma \end{pmatrix} \begin{pmatrix} x_D \\ x_\Gamma \end{pmatrix} = \begin{pmatrix} f_D \\ f_\Gamma \end{pmatrix}.$$

[2]http://www-users.cs.umn.edu/~karypis/metis/main.shtml

Here $A_{D\Gamma}$ represents the sub-domain to interface coupling seen from the sub-domains and $A_{\Gamma D}$ the interface to sub-domain coupling seen from the interface. Since the basis functions from different sub-domains have disjoint support, A_D is block diagonal matrix with diagonal blocks giving the stiffness matrices in the sub-domains (for finite element space associated with the sub-domain with Dirichlet zero boundary condition on Γ). We apply block Gauss elimination to get the following system for the interface unknowns x_Γ:

$$(A_\Gamma - A_{\Gamma D} A_D^{-1} A_{D\Gamma}) x_\Gamma = f_\Gamma - A_{\Gamma D} A_D^{-1} f_D. \tag{1}$$

The matrix $S \equiv A_\Gamma - A_{\Gamma D} A_D^{-1} A_{D\Gamma}$ is called Schur complement matrix associated with the interface variables x_Γ. Once x_Γ is found x_D can be computed by solving $A_D\ x_D = f_D - A_{D\Gamma}\ x_\Gamma$.

Some of the properties of S are as follows. If A is nonsingular then so is S and if A is symmetric and positive definite then so is S. If we denote by $R_\Gamma : R\Gamma x = x_\Gamma$ the restriction operator to Γ we have the identity $S^{-1} x_\Gamma = R_\Gamma A^{-1} (0,\ x_\Gamma)^T$. This suggests thata preconditioner for S may be based on solution techniques with the matrix A. We note that the use of preconditoners for S is mandatory since it's ill-conditioned with condition number bounded by $C\ h^{-1}$ for both 2 and 3-D problems.

A domain decomposition implementation should have sub-domain data structures that provide easy access to the interface nodes (nodes on Γ). For parallel computations the sub-domains should be relatively independent. These requirements are met by keeping the nodes of a particular sub-domain on one processor. Also, every sub-domain keeps its "version" for the nodes on Γ and where the corresponding other "versions" are. The implementation is done in an object oriented style, using *C++*. The main classes, their hierarchy, data members and methods are given on Figure 3.

Class *Subdomain*, as seen from Figure 3, is on the *top* of the class hierarchy. One object of this class may be considered and used as a stand-alone finite element/volume solver. This independence and functionality is obtained through the functionality of the class *Method*, which *Subdomain* inherits. *Method* keeps a finite element *Mesh* (through class *Mesh*) and necessary stiffness matrices (trough class *Matrix*). These matrices, depending on the solved problem, are generated by corresponding functions in *Method*. Other functions that *Method* provides are different solvers (such as CG, PCG and GMRES), preconditioners (multigrid, hierarchical multigrid), a posteriori error indicators and error computing functions (in case the exact solution is known). In terms of domain decomposition such functionality of *Subdomain* is useful because of the following : provides efficient local solvers (for example actions with A_D^{-1} for the Schur complement matrix); easy to map on one processor; readable and easy for further extensions data structure. Besides the stand-alone feature given by its inherited classes, *Subdomain* has specific data fields that complete its functionality as a stand-alone unit in a domain decomposition method. These specific data fields are used to define the connections between the different sub-domains. Their description is as follow :

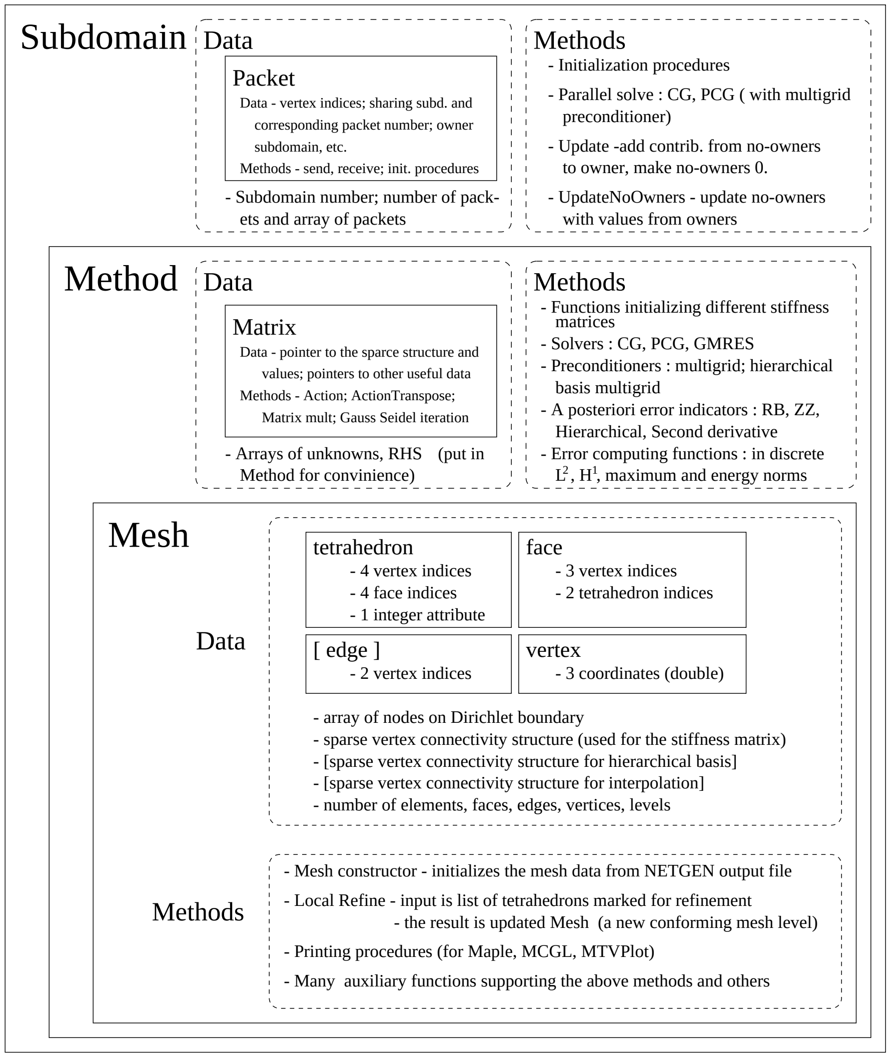
Subdomain
Data
Packet
Data - vertex indices; sharing subd. and corresponding packet number; owner subdomain, etc.
Methods - send, receive; init. procedures
- Subdomain number; number of packets and array of packets
Methods
- Initialization procedures
- Parallel solve : CG, PCG (with multigrid preconditioner)
- Update -add contrib. from no-owners to owner, make no-owners 0.
- UpdateNoOwners - update no-owners with values from owners
Method
Data
Matrix
Data - pointer to the sparce structure and values; pointers to other useful data
Methods - Action; ActionTranspose; Matrix mult; Gauss Seidel iteration
- Arrays of unknowns, RHS (put in Method for convinience)
Methods
- Functions initializing different stiffness matrices
- Solvers : CG, PCG, GMRES
- Preconditioners : multigrid; hierarchical basis multigrid
- A posteriori error indicators : RB, ZZ, Hierarchical, Second derivative
- Error computing functions : in discrete L², H¹, maximum and energy norms
Mesh
Data
tetrahedron
- 4 vertex indices
- 4 face indices
- 1 integer attribute
face
- 3 vertex indices
- 2 tetrahedron indices
[edge]
- 2 vertex indices
vertex
- 3 coordinates (double)
- array of nodes on Dirichlet boundary
- sparse vertex connectivity structure (used for the stiffness matrix)
- [sparse vertex connectivity structure for hierarchical basis]
- [sparse vertex connectivity structure for interpolation]
- number of elements, faces, edges, vertices, levels
Methods
- Mesh constructor - initializes the mesh data from NETGEN output file
- Local Refine - input is list of tetrahedrons marked for refinement
- the result is updated Mesh (a new conforming mesh level)
- Printing procedures (for Maple, MCGL, MTVPlot)
- Many auxiliary functions supporting the above methods and others

Figure 3: Code structure. Main classes along with their data members and methods.

- `SN` – the sub-domains are numbered starting from 0. Every object of class *Subdomain* keeps its number in `SN`;
- `NPackets` – nodes on Γ, shared by the same sub-domains, are grouped into *Packet*s. `NPackets` is integer giving how many packets are defined for *this* sub-domain;
- `Packets` – array of packets for *this* sub-domain (their number is given by the previous field).

The idea of grouping the nodes on Γ into packets is to define the different types of communication between the sub-domains. Such preprocessing is necessary in order to speed up the procedures of sending/receiving data between the sub-domains. The benefits of such structure are explained in our parallel computations Section 6. More precise definition of the *Packet*'s data fields follow :

- `Vertices` – array of the indices of the nodes belonging to the packet. Their number is stored in variable `NPoints`;
- `Subdomains` – array of the indices of the sub-domains that share the nodes in *this* packet. The size is given in `NSubdomains`;
- `PacketAddress` – array of `NSubdomains` integers. Every sub-domain from the array `Subdomains` has its own "version" of *this* packet (according to its local vertex indexing). `PacketAddress[i]` gives the packet number ot that "version" in sub-domain `Subdomains[i]`;
- `Owner` – one of the sub-domains sharing the nodes in *this* packet is defined as owner and this field gives its number. The packet nodes in owner sub-domain are taken as degrees of freedom, the others are considered as "slave" nodes.

Having such data structure for the nodes on Γ simplifies the following common to domain decomposition operations : global matrix actions; global vector operations; local on Γ matrix actions (for example with A_Γ from the Schur complement method); local on Γ vector operations. The utilization of the global operations is shown in the parallel Section 6. The same could be applied for the local on Γ vector operations. We finish this section with demonstration of how local (on Γ) matrix actions are implemented. The example will be for A_Γ from the Schur complement method. We will also use the sparse storage format for the stiffness matrix A, which is given on figure 4. Every sub-domain stores

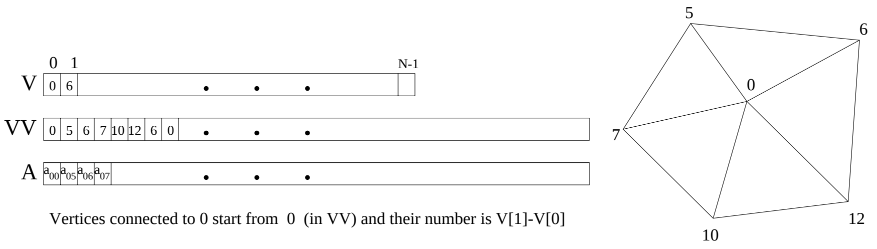
0 1
N-1
V
0 6
VV
0 5 6 7 10 12 6 0
A
a_{00} a_{05} a_{06} a_{07}
Vertices connected to 0 start from 0 (in VV) and their number is V[1]-V[0]
5
6
0
7
10
12

Figure 4: Matrix sparse storage format explained for simplicity with a 2-D example.

a stiffness matrix in the format given on figure 4. The matrix corresponds to stiffness matrix for the corresponding sub-domain with 0 Neumann boundary on Γ. Thus a global action is action on the sub-domains plus adding the contributions from slave packets to the owner (see Section 6). Using this global action we may get $A_\Gamma x_\Gamma$ by computing $A\begin{pmatrix} 0 \\ x_\Gamma \end{pmatrix} = \begin{pmatrix} \dots \\ A_\Gamma x_\Gamma \end{pmatrix}$. Since we don't need the first part of the resulting vector we don't do the full matrix action. We have the nodes on Γ so using the sparse storage structure we can get $A_\Gamma x_\Gamma$ by multiplying $\begin{pmatrix} 0 \\ x_\Gamma \end{pmatrix}$ only with the rows corresponding to nodes on Γ. Similarly one may get efficient action implementation for the prolongation and restriction matrices $A_{D\Gamma}$ and $A_{\Gamma D}$.

5 Adaptive grid refinement

The behavior of the physical process is greatly affected by local properties (coefficients, sources, and boundary data) as well as the singularities due to corners, boundary layers or nonlinear behavior. For such cases it is essential that the numerical method has capabilities to resolve the local behavior of the solution. In the context of the finite element method there are two main techniques for the error reduction. One is based on increasing the order of the algebraic polynomials used in the approximation process, the so called "p – version" of the finite element method (p–refinement), while the other uses polynomials of the same degree, but adaptively refines the grid (by decreasing the mesh size h), the so called "h–version" adaptive refinement (h–refinement).

Our work is mostly concentrated on the h–version of the finite element method. We worked, implemented, and tested (both for 2 and 3-D problems) three error indicators based on the h–version of the finite element method, namely residual based refinement (first introduced by Babuska and Rheinboldt [3] and developed by Becker *et. al.* in [5], [6], and Verfürth [16]), Zienkiewicz-Zhu technique (see [17, 18]), and hierarchical refinement (see [4]). However, we had to adapt the existing technique of a posteriori error estimators for the finite volume element method.

The mesh refinement algorithms are very important since the consecutive refinements have to be done in a way such that the element shapes don't degenerate. Our adaptive mesh generation is based on the bisection algorithm (see, e.g. Arnold et all. [2]). The

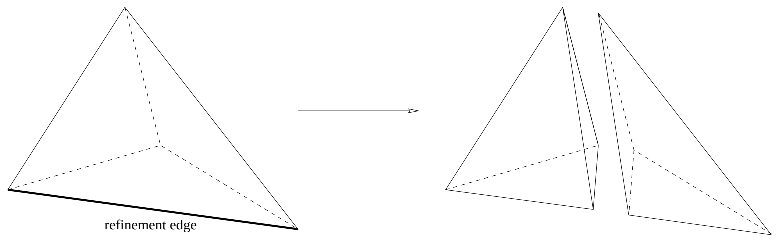

Figure 5: Tetrahedron bisection.

main step in this algorithm is tetrahedral bisection (see figure 5), where a key element is the careful choice of the edge for bisection (called *refinement edge*). The chosen algorithm features data structure that simplifies both the selection of refinement edge and the recursive refinement to conformity once some tetrahedra have been refined. And most importantly, repeated application of the algorithm leads to only finitely many tetrahedral shapes, i.e tetrahedra shape cannot degenerate as the mesh is refined.

The residual method is based on equilibrating certain residuals over the elements. In general, the residuals are obtained by substituting the approximate solution into the problem's weak formulation. Since the approximation is not smooth the residual over one element is composed of two parts. The first part is contribution from the element and the other one is contribution from jumps on the element's boundary. All these quantities are computable through the solution obtained on the current grid. The local grid refinement strategy aims to reduce the finite element size so that the local residuals are equilibrated within a given tolerance criterion.

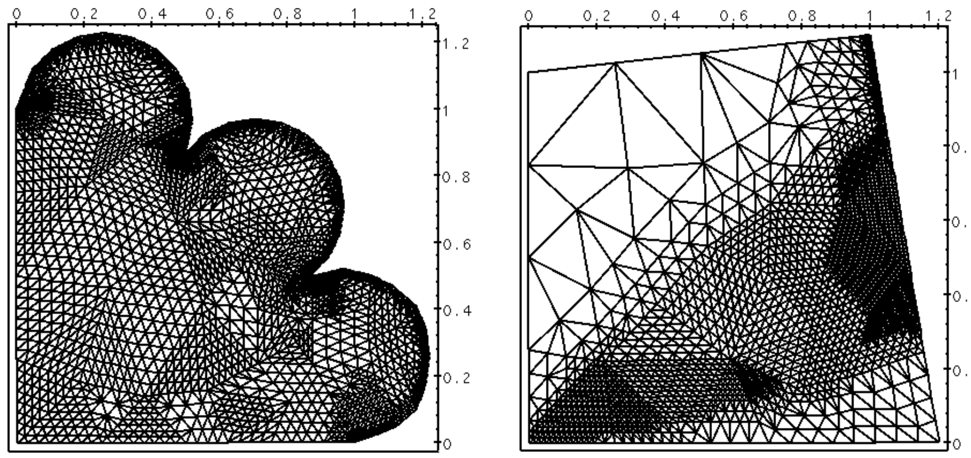

Figure 6: Right: the mesh with 2972 nodes for solving the equation $-\Delta u = 1$ with homogeneous Dirichlet boundary data; Left: the mesh for the convection-diffusion equation $-10^{-3}\Delta u + 2u_x + u_y = 0$ with 1761 grid points

On Figures 6 and 7 we present the results for two examples of such refinement procedure, applied to two 2-D and two 3-D problems. The left Figure 6 shows the mesh obtained for the Poisson equation $-\Delta u = 1$ with homogeneous Dirichlet boundary data. The right Figure 6 shows the mesh obtained for the following convection dominated problem: $-10^{-3}\Delta u + 2u_x + u_y = 0$ in Ω and Dirichlet boundary data so that the left and on the upper edges of Ω the solution is zero while on the rest of the boundary the solution is equal to 1. The solution develops a boundary layer at the upper 2/3 of the right edge of Ω and an interior layer along the line $x = 2y$. Also, there are two corner singularities at the origin and at the upper right corner due to the discontinuity of the Dirichlet data. As seen from the figures the error estimator produces finer grids in the regions where the solution has a boundary layer or singularity. The computed solutions are monotone so no oscillations due to the numerical approximation are produced.

6 Parallel computations

This section describes the machinery that we used and developed for writing parallel finite element software. We start with some fundamental scalar code optimization concepts associated with modern parallel architectures (subsection 6.1). We discuss and

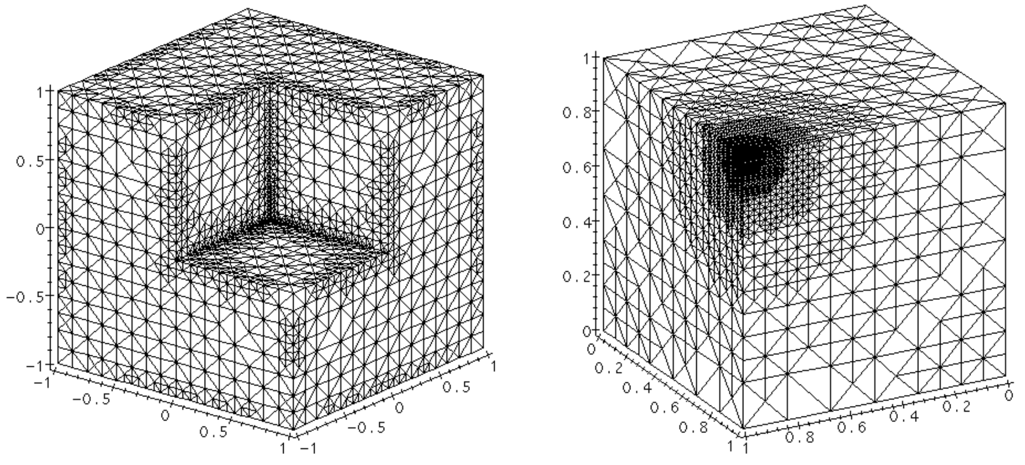

Figure 7: Computational meshes obtained by local refinement due to a corner singularity (left) and delta function source term (right). Both meshes are obtained after six levels of refinement and have $12,350$ and $11,770$ nodes, correspondingly.

give how these may help in a finite element code. The provided material may serve for yet another motivation of why to use *Domain decomposition* techniques. The next two subsections (6.2 and 6.3) give some key concepts in implementing parallel domain decomposition. Also, they may serve as an introduction in writing parallel finite element software under correspondingly shared and distributed memory paradigm.

All illustration and examples of how to use *OpenMP/MPI* are for *C/C++* applications. The included examples were run on a *SGI Origin 2000* system with 8 Mips R10000 processors running at 250MHz, 4MB L2 cache each. Performance values are limited to the same machine. Important about the architecture of such systems (distributed memory *cc-NUMA*) is that they are composed of node cards, which in turn are composed of 2 R10000 processors. The main memory is distributed across the node cards and although the memory is accessible to and shared by all the processors, the latency for memory access by a processor to memory on the node board on which the processor resides is less than the latency for memory access by a processor to memory on a node board different from the one on which the processor resides. This is from where the abbreviation *NUMA* (non-uniform memory access) comes.

6.1 Code optimization

In order to write efficient parallel programs one should first know how to achieve efficiency on a single processor. There are two main concepts, whose understanding is vital for writing efficient scalar code. These are *Locality of reference* (important in improving memory performance) and *Software pipelining* (important in improving the *CPU* performance). We will discuss both of them, but the stress will be on the first one since it's more critical and less machine dependent. As we stated at the beginning we consider

SGI Origin 2000.

6.1.1 Locality of reference

The importance of the concept *Locality of reference* (or simply the rule "Keep things that are used together, close together") arises from the *memory hierarchy*, and more precisely from the different times of accessing data on the different levels in this hierarchy. For example, to access the fastest memory (the registers) it takes 0 *CP*s (clock period or cycle), $L1$ cache 3 *CP*s, $L2$ cache 6 *CP*s, and the main memory $\approx$ 200 *CP*s. The mechanism of something entering the case is as follows. When the program refers to data that is not in the cache the CPU requests a load of a *cache line*, which is a block of 128B (the L1 cache line is 32B), i.e. refering an element from array of doubles will bring a block of 16 elements in the L2 cache and 4 into the L1 cache. Due to the *pipelining* architecture (explained below) the CPU can often continue working while accessing data from the main memory, but still multiple successive cache misses can bring the effective work to a halt because of waiting for data. That's way, for example, a simple computation like

```
A[i][j][k] += B[i][j][k] * C[i][j][k]
```

executed in loop order $i, j, k = 0..100$ is more that 10 times faster than the same computation but performed in loop order k, j, i.

Programs that effectively apply the principle *Locality of reference* are sometimes called *cache friendly*. Except loop interchange, depending on the application, there are many possible techniques to develop *cache friendly* programs. Some of the possibilities are : split computations over large datasets into computations over "data blocks" that entirely fit in the cache memory (technique called *blocking*); group frequently used data fields into single object so they tend to stay in the cache; avoid searching linked lists (especially of big objects); use *memalign()* to allocate important objects on the 128B boundaries and so on.

Developing *cache friendly* finite element software, when iterative solvers are used, is important. Some of the most popular techniques are

- *vertex reordering* – the locality is increased by reordering the vertices. A simple algorithm, which we implemented is *Cuthill-McKee* (see [13]);
- *blocking* – as we mentioned above this is technique where the data set is split into blocks, so that the blocks fit into the cache. For parallel computations, when applying *divide-and-conquer* technique, the cache used on one processor for solving a local problem is greatly utilized. This technique is another motivation for using domain decomposition for parallel computations;
- *fusion* – technique where multiple loops are merged into one. For example, in the *PCG/CG* routine a sequence $z = Ad$; $\beta = z \cdot d$; is better to be merged into one loop, making re-use of the array z.

The second technique is shown to improve the cache memory use in Section 6.3 where we test parallel *CG* method and get *superliner speedup*. This is a term used when the *speedup* using p processors, denoted and defined as $S_p = \frac{\text{exec time on 1 processor}}{\text{exec time on p processors}}$, is bigger than p. The other two techniques, in a test problem of size $200,000$ unknowns using the *CG* solver, gave a speed improvement from 864 seconds to 742 seconds (this includes the time for reordering).

6.1.2 Software pipelining

The other main concept in code optimization, *Software pipelining* (SWP), is important in improving the CPU performance. Such improvement is machine dependent and in R10000 is possible because some of its functional units are pipelined. A pipelined unit is unit partitioned into independent hardware subunits, each specializing a specific phase (to be executed in one *CP*) of the operation that the unit performs. Thus, a unit partitioned in 4 will finish a pipelined sequence of n operations in $n + 3$ *CP*s, compared to $4n$ *CP*s for non-pipelined operations. Now, SWP tries to find a valid rearrangement of instructions (from the **innermost loops only**) so that to engage concurrently as many pipelines as possible. SWP is carried out by the compiler and is enabled if compiling options **-r10000 -O3** are used. The use of SWP in computing `A[i][j][k]+=B[i][j][k]*C[i][j][k]`, $i, j, k = 0..100$ increased the CPU performance from 13 Mflops to 1115 Mflops, making it ≈ 100 times faster.

Some of the cases when SWP looses efficiency are : data dependences occur; long loops; branching (function calls, goto and if-else statements); low iteration counts. Possible techniques to improve SWP are

- *inlining* – in general one inlines only small functions which are called many times within loops, for example, inner-product, vector addition/subtraction functions;
- *splitting/fusing* – splitting wide loops may sometimes enhance prospects for SWP; fusing small loops provides a richer variety of instructions to schedule efficiently with SWP (example for applying in FE software was given in the 6.1.1);
- *outer loop unrolling* – provides richer variety of instructions in the innermost loop (done by increasing the step in the outer loop and explicitly writing the missed iterations in the inner loop); May be combined with *prefetching* – assigning frequently used in the inner loop array element to local variable (done in outer loop) in order to benefit from register reuse; Prefetching is a very often used technique.

6.1.3 Performance monitoring

Tools for performance monitoring are very useful in code optimization. Optimization efforts should be spent on most time consuming routines. **ssrun** may be used to get information on the total amounts of time a program spends on the different routines. To monitor program **fem** compile without optimizations and start with

```
> ssrun -fpcsampx fem
> prof -lines fem.fpcsampx.m####
```

where #### stands for process ID. The output for the case when no *fusing* or *inlining* (for vector addition and inner-product) is done looks like

```
 [index]      secs    %    cum.%   samples  function (dso: file, line)

     [1]    85.497  58.8%  58.8%     85497  Matrix::Action(double*,doubl
     [2]    12.411   8.5%  67.3%     12411  Method::vvadd(int,double*,do
     [3]    11.567   8.0%  75.3%     11567  Method::inprod(int,double*,d
     [4]     7.998   5.5%  80.8%      7998  Method::Convection_LM(float*
     .   .   .
```

To see the values of various hardware performance indices use

```
> perfex -a -x -y fem
```

The output includes statistics for performance indices such as cache line reuse, cache misses, loads/stores, issued instructions, Mflops reached and so on. For more information on how to use see the corresponding **man** pages.

More information about code optimization can be found in [11].

6.2 Parallel FEM using *OpenMP*

OpenMP is a standard agreed upon by major hardware and software vendors. It consists of a portable set of compiler directives, library routines and environment variables that can be used to specify shared memory parallelism in Fortran and C/C++. Advantages of using *OpenMP* are that it's simple, portable and has flexible interface. Unfortunately, knowing *OpenMP* is not enough to write scalable parallel programs. Very often knowledge about the machine is needed in order to perform a proper optimization. We saw how important the optimization issue is in 6.1. It's importance for parallel computations is even greater. For example, it's reasonable to expect that the following fragment of Matrix-vector multiplication ($y = Ax$) is scalable.

```
#pragma omp parallel for private(j, end)
for(i=0;i<dimension;i++){
  end  = V[i+1];
  y[i] = 0;
  for(j=V[i]; j < end; j++)
    y[i] += A[j]*v1[VV[j]];
}
```

In practice, tests on SGI Origin 2000 machines (NUMA architecture) show that the parallel execution may be even slower than the consecutive! One of the problems is that the used arrays are not properly distributed among the processors, which is obviously crucial for the performance of a NUMA architecture machine.

The problem about the data distribution for NUMA machines is solved on software [3] level by introducing new pragmas (`#pragma distribute` and `#pragma distribute_reshape`). Both pragmas are used to allocate the memory for an array among the local memories of the processors. The first one, called *regular distribution*, is constrained to distribution in terms of pages (16KB) and is useful only if each processor's portion of the array is substantially larger then the page size. The second one, called *reshaped*, overcomes the page-level constraints for the cost of certain restrictions on the usage of the reshaped arrays. Once distributed, the arrays may be redistributed (`#pragma redistribute`). For more information on the subject (syntax and so on) see [14], Chapter 5.

Having discussed the crucial issue of data distribution we concentrate again on the development of parallel FE software. The easiest way to parallelize the FE code, using the discussed until now techniques, is to use the sequential one and add some parallel constructs. The algorithm is as follows. First, parallelize the loops using the *OpenMP* directives and functions. Second, increase the locality by some vertex reordering technique. Finally, improve the data distribution. Such strategy looks feasible. The practical implementation shows that the appeal is only theoretical. The problems are :

- *inefficient memory usage* – data distribution can be applied only to statically allocated arrays (fixed size), whereas in an adaptive finite element the memory allocation is usually dynamic. To overcome this problem one may have to allocate "bigger" arrays and dynamically redistribute them on each refinement level using *block-cyclic* distribution with blocks of size $\left\lceil \frac{\texttt{\# nodes}}{\texttt{\# processors}} \right\rceil$. Also, in an adaptive FE one would like to make use of the different refinement levels by defining multi-level preconditioners, which could further worsen the memory usage inefficiency;

- *poor scalability* – due to many synchronization points. Also, in Matrix-vector product ($y = Ax$) even in a perfect data distribution one processor still have to access parts of x residing on the local memories of other processors. We also suspect that there is computational overhead using reshaped arrays;

- *programming efforts* – the use of reshaped arrays requires change of the syntax of many functions, more precisely, it requires specific format for passing such arrays as function arguments.

Based on the mentioned problems and our computational experience, the conclusion is that data placement is unlikely to help for direct parallelization of finite element code and redesign of the algorithm is needed in order to get efficient, well scaled software. The choice is of course the use of domain decomposition. The proposed data structure is easy to use on a shared memory machine. Only basic *OpenMP* directives and functions

[3] On hardware level there is the so called *page migration*, which is automatic reallocation in the main memory of whole pages (16KB)

are used. The code looks like a translation of the *MPI* version. For example, the subdomain initialization in the main function, using *OpenMP* (compared to *MPI*, see the next subsection) looks like :

```
#include <omp.h>
. . .

Subdomain **S;

main(){
  int np;                              // Number of available processors
  cout << "Insert the number of processors : "; cin >> np;

  cout << "\nInput the file name : "; cin >> fname;
  Method m(fname, 0, 0);               // Initialize mesh, where fname is
                                       // the output from Netgen.
  int *tr;
  tr = new int[m.GetNTR()];
  m.DomainSplit(np, tr);               // Split the domain into "np" domains
                                       // using Metis.
  S = new PSubdomain[np];              // For simplicity we take S :
                                       // Subdomain **S; to be global.
  omp_set_num_threads( np);            // Set the number of threads
  #pragma omp parallel                 // Parallel region - to initialize
  {                                    // the sub-domains
    int myrank;
    myrank = omp_get_thread_num();     // Get the thread number

    S[myrank] = new Subdomain(myrank, &m, tr);
  }
  . . .
}
```

We don't use the complicated *OpenMP* data distribution constructs. Instead we use the so called *first touch* rule. This is a rule that memory is allocated to the processors which are the first to access or touch the data. In the example above processor with thread number `myrank` allocates in its local memory sub-domain with index `myrank`. The communication between the sub-domains is done using the shared memory. For example in implementing a global action, instead of non-owners sending contributions to owners (next sub-section) here the owners directly take and add the slave contributions. This is done after the non-owner have prepared the packages (as in the *MPI* case) in consecutive addresses, so that cache reuse is achieved. For dot products we use `#pragma omp critical` to add the sub-domains' dot products. The other alternative, the *OpenMP* `reduction`, seems computationally not efficient. To use the *OpenMP* functions one have to include file `omp.h`. The compilation should be with `-mp` option.

6.3 Parallel FEM using *MPI*

Message-Passing Interface or *MPI* is a complex system of 129 functions which allow distributed memory parallel programing model. In such model the computation comprises one or more processes that communicate by calling library routines to send and receive messages to other processes. Although there are many *MPI* functions usually 5 or 6 are enough. We will describe these main functions and show how they are used in our code. The examples will be how the data structure was used in a parallel implementation of *CG* using *MPI* (in C++). A simple introduction model may be demonstrated with a fragment of our main function.

```
#include <mpi.h>                          // Has to be included

int main(int argc, char *argv[]){
  char fname[100];
  int Refinement;

  int np;
  cout << "Insert the number of processors : "; cin >> np;

  cout << "\nInput the file name : "; cin >> fname;
  Method m(fname, 0, 0);

  int *tr;
  tr = new int[m.GetNTR()];
  m.DomainSplit(np, tr);

  int myrank;
  MPI_Init(&argc, &argv);                 // Initiate an MPI computation
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);// Determine the process identifier
                                          // (in myrank)
  Subdomain dd(myrank, &m, tr);           // Initialize sub-domain ‘‘myrank’’
  dd.Solve();                             // Parallel solve

  MPI_Finalize();                         // Terminate the computation
}
```

Another function, used in implementing dot-product, is `MPI_Allreduce`. `Solve` from the last example is called from all the processes. In `Solve` we have a call to a standard `CG`. The only change in the consecutive dot product in this `CG` routine is that at the end we add the following :

```
  . . .                         // compute res = (x, y)
  #ifdef DOMAIN_DECOMPOSITION
    double total;
    MPI_Allreduce(&res, &total, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
    return total;
  #else
    return res;
  #endif
```

The meaning of the arguments is what intuitively one would guess looking at the example. The other vector operations (addition/subtraction, scaling) are as in the consecutive version. The other important *MPI* functions, sending and receiving, will be demonstrated in the realization of a parallel global Matrix-vector product ($y = Ax$).

```
void Matrix::Action(double *x, double *y){
  #ifdef DOMAIN_DECOMPOSITION         // The owners update the values of x
  Subdomain->Update_Slave_Values( x); // in the slave nodes
  #endif
  . . .                               // standard  y = A x
  #ifdef DOMAIN_DECOMPOSITION
  Subdomain->Update( y);              // Slave nodes add their values
  #endif                              // (contributions) to the owners
}
```

Functions `Update_Slave_Values` and `Update` have similar implementation and they use the same *MPI* functions, so we will show only function `Update`. This function updates the values of its argument by adding the contributions from no owners to owners. It also makes the values at the no-owned nodes 0.

```
void Subdomain::Update(double *x){
  int i, j, k, PNum, n = 0, start;
  double Buf[MAX_PACKET], Bdr[6*MAX_PACKET];

  MPI_Request request;
  MPI_Status Status;

  for(i=0; i<NPackets; i++)
    if (Packets[i].Owner != SN){
      start = n;
      for(j=0; j<Packets[i].NPoints; j++){    // Put in Bdr all the infor-
        Bdr[n++] = x[Packets[i].Vertices[j]]; // mation that has to be sent.
        x[Pa[i].Ve[j]] = 0.;
      }
      // Send to the owner Packets[i].Owner - the corresponding packet
      // number is Packets[i].Pack_Num_Owner (see the receiving part)
      MPI_Isend(&Bdr[start], Packets[i].NPoints, MPI_DOUBLE, Packets[i].Owner,
                Packets[i].Pack_Num_Owner, MPI_COMM_WORLD, &request);
    }

  // The owners receive packets from neighboring subdomains and do the
  // necessary corrections.
  for(i=0; i<NPackets; i++)
    if (Packets[i].Owner == SN){              // Every owner should read
      for(j=0; j<Packets[i].NSubdom; j++){    // Pa[i].NSubdom packets
        MPI_Recv( Buf, MAX_PACKET, MPI_DOUBLE, MPI_ANY_SOURCE,
                  MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
        MPI_Get_count( &Status, MPI_DOUBLE, &n);// how many were read
        PNum = Status.MPI_TAG;

        for(k=0; k< Packets[PNum].NPoints; k++)
          x[Packets[PNum].Vertices[k]] += Buf[k];
      }
    }
}
```

We put in `Bdr` all the information that has to be send in order to use non-blocking send (function `MPI_Isend`). The alternative, using blocking send, is slower since the sending procedure `MPI_Send` doesn't complete until the whole message is delivered. A possibility of deadlock is possible in this case.

The header file `mpi.h` has to be included. The compilation should be with `-mpi` option. The executable is started with `mpirun -np #_of_processors executable`. More about the syntax of the used and other *MPI* functions can be found in [8].

With the so implemented parallel CG method we get *superliner speedup*. As we mentioned before this is a term used when the *speedup* using p processors, denoted and defined as $S_p = \frac{\text{exec time on 1 processor}}{\text{exec time on p processors}}$, is bigger than p. Such phenomena is possible and is explained with the efficient cache use due to reducing the problems by the splitting of the original domain. The demonstration results are from solving $-\triangle u = 1$ with homogeneous Dirichlet boundary data on a rectangular parallelepiped of size $1000 \times 500 \times 500$. The results are summarized in table 6.3.

problem size	Time in seconds/speedup			
	1 processor	2 processors	4 processors	8 processors
6,481	1.00	0.53	0.34	0.22
		1.90	2.92	4.52
17,913	2.98	1.33	0.73	0.52
		2.24	4.08	5.73
43,293	13.52	5.78	3.12	1.64
		2.34	4.33	8.24
69,910	31.75	14.95	6.37	3.32
		2.12	4.98	9.56
447,972	699.7	338.7	159.7	75.5
		2.07	4.38	9.27

Table 6.3. Computational results for the time and the corresponding speedups for different problem sizes. We solve $-\triangle u = 1$ with homogeneous Dirichlet boundary data on a rectangular parallelepiped of size $1000 \times 500 \times 500$. Problem size is the number of degrees of freedom.

References

[1] M. Ainsworth and J. T. Oden: A unified approach to a a posteriori error estimators based on element residual methods. Numer. Math., **65** (1993) 23-50

[2] D.N. Arnold, A. Mukherjee, and L. Pouly: Locally adapted tetrahedral meshes using bisection.

[3] I. Babuska and W. C. Rheinboldt: Error estimates for adaptive finite element computations. SIAM J. Numer. Anal., **15** (1978) 736-754

[4] R. E. Bank and R. K. Smith: A posteriori error estimates based on hierarchical bases. SIAM J. Numer. Anal., **30** (1993) 921-932

[5] R. Becker, C. Johnson, and R. Rannacher: Adaptive error control for multigrid finite element methods. Computing, Springer-Verlag, 1995

[6] R. Becker and R. Rannacher: Weighted a posteriori error control in finite element methods. Preprint 96-1 (SFB 359), Proc. ENUMATH - 95, Paris, 1995

[7] U. Elsner: Graph Partitioning: A Survey. Technical Report SFB393/97-27, TU Chemnitz, 1997

[8] W. Gropp: Tutorial on MPI: The Message-Passing Interface. Mathematics and Computer Science Division, Argonne National Laboratory (available on Internet [4])

[4]http://www-unix.mcs.anl.gov/mpi/tutorial/index.html

[9] G. Karypis and V. Kumar: Multilevel k-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed Computing, 48(1):96-129, 10 January 1998

[10] G. Karypis and V. Kumar: A fast and highly quality multilevel scheme for partitioning graphs. SIAM Journal on Scientific Computing, Volume 20, Number 1 pp. 359-392,1998

[11] D.A. Padua and M.J. Wolfe: Advanced Compiler Optimizations for Supercomputers. Communications of ACM (special issue), vol. 29, no. 12, pp. 1184-1201, Dec. 1986

[12] A. Quarteroni and A. Valli: Domain Decomposition Methods for Partial Differential Equations. Clarendon Press, Oxford, 1999

[13] Y. Saad: Iterative methods for Sparse Linear Systems. PWS Publishing Company, 1995

[14] Silicon Graphics, Inc.: MIPSpro C and C++ Pragmas. Document Number 007-0701-130 (available on Internet [5]), 1999

[15] Silicon Graphics, Inc.: C Language Reference Manual. Document Number 007-3587-003 (available on Internet [6]), 1999

[16] R. Verfürth: A posteriori error estimators for convection-diffusion equations. Numer. Math., **80** (1998) 641–663

[17] O.C. Zienkiewicz, and J.Z. Zhu: A Simple Error Estimator and Adaptive Procedure for Practical Engineering Analysis. Int. J. Numer. Meth. Engng., **24** (1987) 337-357

[18] O.C. Zienkiewicz, J.Z. Zhu, A.W. Craig, and M. Ainsworth: Simple and Practical Error Estimation and Adaptivity: h and h-p Version Procedures. Adaptive Methods for Partial Differential Equations, SIAM, 1989

[5] http://autarch.loni.ucla.edu/ebt-bin/nph-dweb/dynaweb/SGI_Developer/

[6] http://autarch.loni.ucla.edu/ebt-bin/nph-dweb/dynaweb/SGI_Developer/