Object-oriented structures for Domain Decomposition methods

J. Gopalakrishnan, R. Lazarov, J. Pasciak

Abstract

A set of data structures useful in implementing domain decomposition methods are described. A code that uses these data structures to implement the conjugate gradient method, with the operator evaluations, inner-products and vector additions done in parallel, is described, and its parallel speedup is analyzed. Possible extensions to the code are pointed out.

1 Introduction

This report describes a set of object-oriented structures useful in implementing domain-decomposition methods for solving finite element systems. In particular, it is suited for massively parallel computations of flows in porous media.

To explain the approach, suppose we are required to solve a flow equation (pressure, saturation, concentration etc.) for a given domain subject to various boundary conditions and a source term. The given domain is triangulated into finite elements, and the resulting mesh is split into *sub-domains* of almost equal size and minimal interface (like in figure 1). Domain decomposition deals with methods to solve the system on the whole domain by doing independent computations on separate sub-domains and transfering data between sub-domains when necessary. Since the whole computation is broken up into computationally independent tasks on sub-domains, a sub-domain with its data and functionality, may be associated with one of a number of concurrent processes.

A sub-domain may therefore be thought of as an object processing its own data, and communicating with other sub-domain objects. This, and the structured nature of finite element computations, makes the choice of language as C++ irresistible. Further, since each sub-domain operates on its own data, we may expect to execute the code in distributed memory parallel machines (and certainly shared memory ones).

A code that provides such a sub-domain class and uses it to parallelise the operator evaluations, inner-products and vector additions that occur in the conjugate gradient method is presently available. Since the current code has been written with the objective of demonstrating parallelism of the proposed structures, incorporation of a wide range of physical parameters for input has received lesser priority. In its present state, it can compute the piecewise linear finite

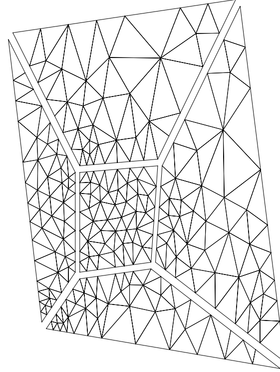


Figure 1: Domain Decomposed

element approximation to the solution of $-\Delta u = f$, with zero Dirichlet or free Neumann boundary condition, on arbitrary triangulations. However, the structures are designed so that extension to more general equations and boundary conditions are easy to accomplish, and this will indeed be the next step in the development of this code. We first explain the data structures and procedures used in the code. Next, conventions used in reading input to the program are given. Finally we report some examples that were run to illustrate the code performance.

2 The code structure

The aim is to solve a finite element problem on a large domain by partitioning the domain into sub-domains, and assigning the sub-domains and their associated data to processors. The computation required to get the finite element approximation need to be reduced to parallel computations on the sub-domains plus some interface communications.

It is also required that the classes constructed be general enough to permit implementation of a variety of Domain Decomposition methods [2, 1].

With these in mind, data structures are designed. To get an overview of the structure of the code refer figure 2 in page 3. Note that figure 2 is only a guideline, and all features there are not as yet implemented.

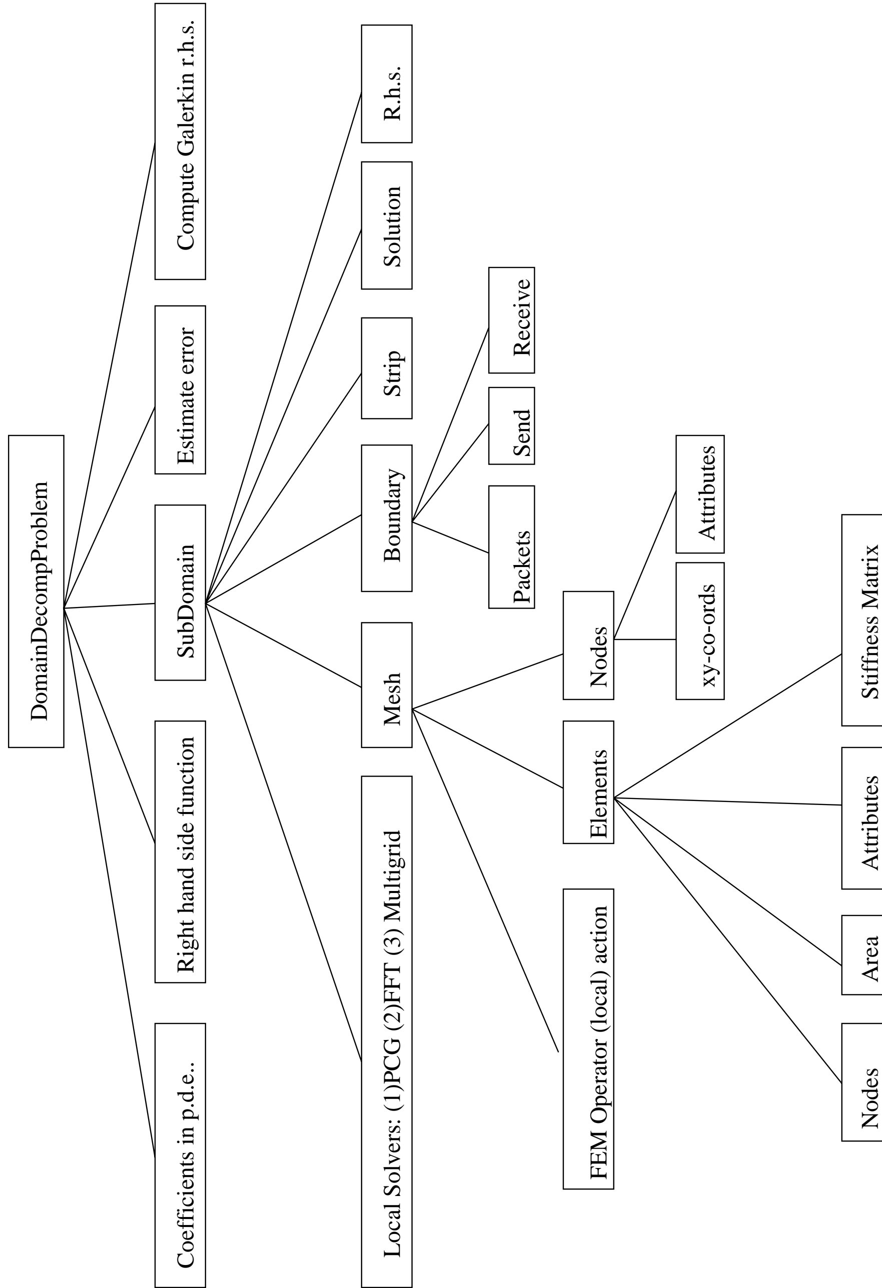


Figure 2: Objects and their dependencies

2.1 Classes of Problems

An abstract Domain Decomposition Problem class "DomainDecompProblem" is constructed, from which various specific problems can be derived (see figure 3), each derived problem differing in the way it will solve the domain decomposition problem.

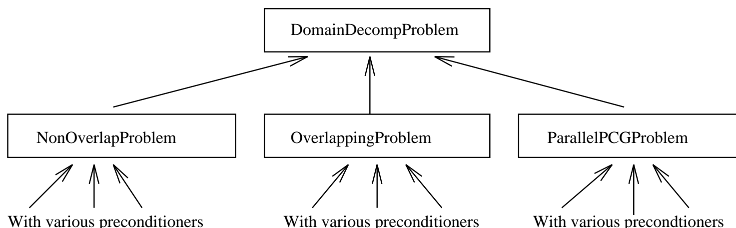


Figure 3: Problem classes

Any object belonging to a Domain Decomposition class must have a definition of coefficient functions and the right hand side function of the partial differential equation to be solved. In the event the right hand side function is known only discretely, the routine that can compute the right hand side of Galerkin equations may be overloaded to work with such discrete data.

"ParallelPCGProblem" is derived from "DomainDecompProblem". It can "Solve()" the Domain Decomposition Problem using Preconditioned Conjugate Gradient iterations, by taking advantage of the fact that Conjugate Gradient can be coded to be run in parallel. Conjugate Gradient is coded as a template which can operate on any structure irrespective of how the structure stores the vector of values iterated (see subsection 2.5). The supporting routines needed by Conjugate Gradient: the finite element matrix - vector multiplications, the l^2 inner-product, scalar-multiplication and additions of vectors, can all be done in parallel locally on each SubDomain and corrected to true global values by communicating with neighboring SubDomains. At run time, each processor calls Conjugate Gradient. The supporting routines (operator evaluation, inner-product, scalar multiplication and addition of vector structures) of each invoked conjugate gradient communicate with supporting routines of conjugate gradients invoked in other processors. The net result is that this parallel process mimics a serial conjugate gradient, i.e., the residual norms, the scaling factor for search direction etc. are all the same as a serial conjugate gradient applied to the problem on the un-partitioned domain. The results should differ only due to computer round-off errors.

however the supporting routines communicate with other processors, so that the net result is that each of the initiated Conjugate Gradient-s behave the same way (residual norms, the scaling factor for search direction etc. are all the same).

Code Extendibility: Another derived class of "DomainDecompProblem" that will be coded in future is called "NonOverlapProblem". This class will have a

different "Solve()". It will implement a more complicated Domain Decomposition algorithm: To find a finite element projection u onto the finite element space S_h, given by $A(u,v) = (f,v) \ \ \forall v \in S_h$ we solve problems on smaller subdomains Ω_i (with corresponding local finite element spaces $S_h^0(\Omega_i) = \{\phi \in S_h : supp(\phi) \subset \bar{\Omega}_i\}$) and correct as follows.

1. Compute u_i on $S_h^0(\Omega_i)$ that solves

$$A(u_i, \phi) = (f, \phi) \ \ \forall \phi \in S_h^0(\Omega_i)$$

2. Compute σ that solves the following problem on the interface *boundaries* created when the domain was broken up.

$$\langle \sigma, \delta \rangle = (f, \bar{\delta}) - \sum_{i=1}^{m} A(u_i, \bar{\delta}) \ \ \forall \delta \in S_h(\Gamma)$$

   where $\Gamma = \partial\Omega - \cup_i \partial\Omega_i$, $S_h(\Gamma)$ consists of restrictions of basis functions with nodes on Γ, $\bar{\delta}$ for a $\delta \in S_h(\Gamma)$ is the trivial extension (by 0) of δ into Ω and $\langle \sigma, \delta \rangle$ form is defined as $\langle \sigma, \delta \rangle = A(\omega_\sigma, \omega_\delta)$, with ω_σ and ω_δ standing for discrete harmonic extensions of δ and σ into Ω.

3. Compute u_h, the discrete harmonic extension of σ computed in step (2) and add it to u_i of step (1)

$$u = u_h + \sum_{i=1}^{m} u_i$$

   to get the final solution.

For this algorithm, usually referred to as the non-overlapping domain decomposition algorithm (or the Schur complement method), it is well-known how to precondition the boundary problem (step 2). The preconditioner will also be implemented. In fact presently a serial code with simulated parallelism exists that implements this algorithm with almost the same data structures as explained here.

Overlapping Domain Decomposition algorithms may be implemented in future by constructing yet another derived class "OverlappingProblem". Note that the definitions of communicating objects "Packets" (see sub-section 2.3) are amorphous enough for it to be acceptable as communicating objects in a possible implementation of overlapping domain decomposition algorithms.

2.2 Sub-domains

A "SubDomain" basically consists of a "Mesh" (see subsection 2.4), a boundary (see 2.3), and an optional Strip. Finite Element structural requirements

are taken care of “Mesh”, while all communication structures are in “SubDomainBdry”.

The member Strip is a collection of triangles which have at least one node on the interface boundaries of “SubDomain” (interface boundary meaning the boundaries created while decomposing the original domain into sub-domains). The constructor for “SubDomain” has the option for leaving Strip unconstructed, as Strip is not a structural necessity for all kinds of “DomainDecompProblem”s.

Code Extendibility: SubDomains can solve Finite Element problems posed on them by using preconditioned Conjugate Gradient. If the SubDomain is square with uniform triangulation, it can solve Poisson's equation faster using Sine Transform. (A Multigrid solver for nested meshes is also in the works.) The ability to do local solves is not utilised by “ParallelPCGProblem”, but will be useful for other problems, if and when they are implemented.

2.3 Sub-domain boundaries and communications

“SubDomainBdry” is a collection of “Packet”s. It can send or receive packets to or from other sub-domains. This is the way all communications are handled. A “Packet” between SubDomains S_i and S_j is a subset of the set of vertices (or rather the set of vertex numbers) that are both in S_i and S_j. It also has space to contain data on those vertices. Note that a node that is shared by S_i and S_j though conceptually a single entity will exist dichotomously in both the processor containing S_i and in the one containing S_j. Necessarily therefore, “Packet”s exist in pairs - a “Packet” of S_i that transmits information to S_j has a mirror image of itself in S_j that will transmit information from S_j to S_i. “Packet”s may therefore be thought of as communication buffers. Declaring the space for buffering right in the data structures of the code, before any communication starts, makes the code safer, in that an exit at run time for lack of buffering space will not be made.

To do the book-keeping of values on vertices that has mirror images on other processors, we group the numbers of those vertices into packets and assign one of the sharing sub-domains as the owner of that packet. The owner will be responsible for maintaining correct values on the vertices in the owned packet and transmitting them to copies when copies need to know the correct value they should be having. For each node i, let P_i be the set of processors (or sub-domains) which have i as a node. The sub-domain S_k will be said to own node i, if

$$k = \min_{j \in P_i} j.$$

The copy of node i residing in sub-domain S_k will then be called an *owner node* and its mirror images in other processors will be called *copy nodes.*

To construct packets of say, sub-domain S_i, group together its owned nodes into packets $Pc(i,j)$, for $j = 0, 1, ..$ as follows: $Pc(i,j)$ contains those nodes which are owned by processor i and are also nodes of processor j (of course, $j > i$), i.e., $Pc(i,j)$ is the collection of owner nodes of S_i that have its mirror copy nodes in S_j. Of course, sub-domain S_j would have among its packets,

a packet mirroring the $Pc(i,j)$ we have just constructed - it would consist of precisely those copy nodes in S_j that mirror the nodes in $Pc(i,j)$. We have presently divided the collection of all owner nodes of sub-domain S_i into packets (let 'divided' not give the wrong idea: one node can indeed belong to more than one packet).The remaining interface boundary nodes of S_j are copy nodes, and they will get split into packets too - packets that mirror packets in other sub-domains who own these nodes.

To see how all this helps in communications, let us restrict ourselves to the distributed memory version of the code, and consider this often used sequence of member functions of "SubDomainBdry":

```
OwnersPostReceive(requests);
CopiesPostSend(x, requests);
CopiesIncrement(x, requests, status);
CopiesPostReceive(requests);
OwnersPostSend(x, requests);
CopiesOverwrite(x, requests, status);
```

This will correct increments made by each domain to vector x to true global increments. Here 'requests' and 'status' are variables required by the Message Passing Interface (MPI). All increments made individually by a sub-domain to the vector x that lives in the same processor as the sub-domain (but is part of a global vector) are loaded onto copy packets. These packets are then transmitted to their mirror owners. There they are received and the local copy of x is incremented. This much is completed when each processor reaches the end of the first three lines above.

The owner packets are now loaded with the true global values that x should have at its vertices that have copies elsewhere. They are now sent to the sharing domains wherein the copy packets receive these values and overwrite the x-values at the proper places with the received values. This much is done by the last 3 lines above.

Data transmissions are made possible through the use of MPI (*Message Passing Interface*) library functions [3, 5]. This ensures code portability across many distributed memory machines. All sends are *ready* sends that on some machines (like Intel Paragon) will avoid some hand-shaking protocols and decrease communication time. This is possible because the code is designed to be able to post (non-blocking) receives before sends and let the processors expect the impending send. All point-to-point communications (i.e., all communications other than collective reductions and broadcasts) are through the above listed routines; every effort has been made not to let the transparency of the structures be garbled by communication calls. No wild card matching at the receive end is ever done. All receives know exactly who the source is. At each point-to-point communication, two pieces of information other than the message data needs to be transmitted: Say, the $i-$th packet P_i of SubDomain S_1 is to be transmitted to its mirror packet listed at the $j-$th position, Q_j in SubDomain S_2. The message should then contain the number j so that at the receiver end

S_2 knows which among all its packets should receive the incoming information. The message should also indicate in some way if the incoming data is to be incremented or overwritten at the receiver end. Both these pieces of information are coded together into one number and that number becomes the message *tag*. A posted receive knows the tag with which its expected message will come with; so no wild card matching is done for tags too. This imposes a small limitation: MPI only guarantees that tags in the range $0..32,767$ are accepted. This for our code translates to saying that a SubDomainBdry cannot have more than $3,276$ packets; hardly a serious limitation.

Implementations of MPI on shared memory architectures are presently becoming available, though some have not been used enough to be rid of bugs. So it may be expected that the version of the code written using MPI may soon be ported even to shared memory machines easily.

The other version of the code that is now being run on shared memory SGI uses a simulated network to communicate. There sub-domain boundaries post "Message"s (which is a defined data type) on an object called "Network" (which is nothing but a linked list of "Message"s) and receive "Message"s from the "Network", whenever there is a need to communicate. 'Posting' and 'receiving' are implemented as appending and extracting from the linked list called "Network". Otherwise, with packets as before, the structure of "SubDomainBdry" is the same as in the version using MPI.

Code Extendibility: An attractive feature of MPI that we have not used is the notion of Virtual Topologies. An application like ours where a communication graph is known before hand may benefit from telling MPI to renumber processes so that the application demands may match the hardware connectivity. This feature can be incorporated easily into the code if a communication graph can be given as input to the program. To make this easy, the MPI communication *context* (a pointer) is made a member of "DomainDecompProblem". The code always uses this member name instead of explicitly naming the MPI communicator. At the time of construction of the problem class it is presently set to the default MPI_COMM_WORLD. It may just as well be set to a communicator with a prescribed virtual topology.

2.4 Mesh

"Mesh" consists of "Triangle"s (Elements) and "Vertex"s (Nodes). Given a vector of values (at nodal positions), it can produce the result of action of the finite element matrix operator on the vector.

"Triangles" are composed of "Vertex"s, and have in addition its element stiffness matrix stored as a member. It is easy to put in more attributes if necessary to these elements (like material properties). Members of "Vertex"s have information about the x, $y-$ co-ordinates of the point it represents (again more attributes may be added to vertices if need arises).

2.5 Conjugate Gradient Template

The Preconditioned Conjugate Gradient Method (PCG) that solves a symmetric positive definite system of linear equations is implemented as a C++ template [6]. There is a provision for incorporating a preconditioner if one is available. The routine takes as input 3 objects of unknown type and a user specification as to whether an estimate of condition number is to be calculated. The first 3 objects conveys to Conjugate Gradient, problem information, initial guess, and right hand side in some way (not explicitly known to the template). The condition number estimation is done with the help of LAPACK library. It is possible to specify at *compile* time that condition number estimations be not done (to make the routine a little more efficient). Two other parameters that also need to be tuned at compile time are: the maximum number of iterations permissible, and error tolerance level below which if successive updates differ, the routine should exit.

To perform the algorithm, the template relies on 4 routines (to solve $Au = b$):

1. Given x compute Ax.
2. Given x, y, compute l^2 inner-product (x, y).
3. Given scalars a, b, and vectors x, y, compute $ax + by$.
4. Given x, compute Bx, where B is a preconditioner to A.

The definitions of these routines overloaded to work with specific data types will be used at an instantiation of the template with those data types.

As a result of this template mechanism, the same code can be used for local sub-domain solves using PCG, for a global parallel PCG solve, or for solving the 'boundary problem' in the Non-overlapping domain decomposition case.

3 Input and Output

Each process collects its own data from a set of 3 files that contains data to build the one and only sub-domain that is associated with that process. Upon finishing computations, each process writes the values of the computed solution on the sub-domain associated with it. In this sense the I/O is parallel.

3.1 Input file names

The input for the program consists of a sequence of files each of which contains data for constructing elements, nodes, or boundary of one sub-domain. For example, *Domain201.ele, Domain201.node,* and *Domain201.bdr* would give data for constructing elements, nodes and boundaries respectively of SubDomain No. 201. The prefixed string (in this case "Domain") can be arbitrary, but should remain the same for all files pertaining to one problem. The number in a file name that follows the prefix identifies the particular sub-domain for which the

data is being provided. The extension specifies if the data pertains to elements, nodes, or boundary of the sub-domain specified by the number in the file name (for example ".ele" specifies that file contains element data). Such a prefix string, an output file name (i.e., a string that will be a common prefix to all output file names), and an integer representing the total number of sub-domains are the only terminal inputs to the program.

3.2 Input File Formats

The ".ele" and ".node" files follow the same format as the Mesh Generator *Triangle* (see the web-page http://www.cs.cmu.edu/ quake/triangle.html).

We adopt the following convention for *boundary markers* (in ".node" files).

```
0  : node not on any boundary (neither the interior nor
     the global boundary)
-1 : node on interior boundaries
-2 : node on global boundary with a Neumann/Robin bc.
-3 : node on global boundary with another Neumann/Robin bc.
-4 : node on global boundary with yet another Neumann/Robin bc.
-5 : and similarly more -ve integers for various
     other Neumann/Robin bc.
1  : node on global boundary with Zero Dirichlet bc.
2  : node on global boundary with some other Dirichlet bc.
3  : and higher +ve integers for various other Dirichlet bc.
```

Note that nodes on interior (or interface) boundaries created when breaking up the domain must be marked -1.

It is assumed that Vertex numbering in a sub-domain starts with 1 (not 0).

The ".bdr" files gives information to construct packets and hence sub-domain boundaries (see 2.3). A ".bdr" file is organised as:

```
#____________________beginning of file______________________
no np
pnd1 add1 lad1 own1
nd#
nd#
nd#
nd#
:
:
:
nd#
pnd2 add2 lad2 own2
nd#
nd#
nd#
:
```

```
:
:
nd#
pnd3 add3 lad3 own3
nd#

nd#
:
:
etc......
#___________________end of file_____________________________
```

where all symbols are integers and have the following meanings:

```
no  = SubDomain Number.
np  = Number of packets for this SubDomain ( SubDomain(no) ).
pnd1= Number of nodes in Packet(1) of SubDomain(no).
add1= Address of Packet(1), i.e, Number of SubDomain where
      Packet(1) will be sent.
lad1= Local address of Packet(1), i.e., the packet number of the
      copy of Packet(1) residing in SubDomain(add1).
own1= Number of the SubDomain that owns Packet(1).
nd#   Generic name for node numbers.
pnd2= Number of nodes in Packet(2).
add2= ..similarly..
lad2= ..similarly..
```

It is assumed that in the list of nodes for a packet and for its mirror image in another sub-domain (these lists will appear in two different files) the numbers of the nodes and those of its mirror images should appear in the same order.

3.3 Output

Each sub-domain writes its computed solution values on a file. These files have names with a common prefix string which is specified at terminal input. The string obtained by concatenating this prefix with the sub-domain number gives the name of the output file of that sub-domain. The solution values at nodal points are written into this file element by element. That is, the file contains output data split into 3 columns: first col for x-values, second for y-values, third for Solution values at (x,y). The first three lines should give the three (x,y,z) sets for the three nodes of the first triangle. The second three lines for the second triangle, and so on. (Obviously there are many repetitions.)

Maple for example, can use such files to plot the solution. (However, to view solutions of really large problems, *Maple* is not be used, as the worksheet that plots values on general triangulations is clumsy and inefficient.)

The program reports relative errors (if true solution is known) in max-norm and l^2-norm.

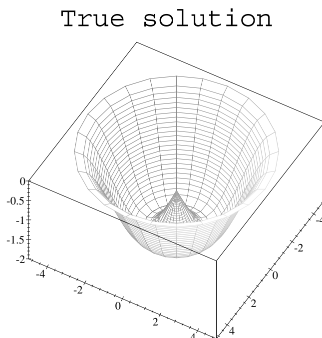
It also reports condition numbers of problems solved through Conjugate Gradient if a request for condition number estimation was given to Conjugate Gradient (and if LAPACK libraries were successfully linked).

4 Test Runs

All runs are made on a distributed memory 28-node Intel Paragon. Some are made also on a shared memory 24 CPU SGI Power Challenge. The Paragon is capable of doing 100 MFLOPS per float (and 80 MFLOPS per double), while the SGI is capable of 400 MFLOPS per float.

4.1 A Dirichlet Problem

P.D.E: $-\Delta u = \frac{cos(r)}{r} - sin(r)$. Note that the singularity on the right hand side is integrable.
Domain: Circle centered at 0 of radius $3\pi/2$.
Boundary condition: $u = 0,$ on the circle of radius $3\pi/2$.
True Solution: $-(1 + sin(r))$. Note the kink at origin.

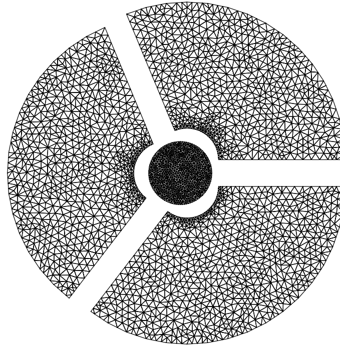


Mesh: has 3006 vertices and 5511 triangles. It is refined on a circular region near the center, in the hope of recovering the kink of the true solution nicely.
Sub-Domain splitting: The refined circular region near the center forms one sub-domain. The remaining annular region is split into 3 equal angular sectors. The total number of triangles gets split between sub-domains as

$$1272 + 1425 + 1360 + 1454 = 5511$$

and the collection of vertices gets divided as (some are shared)

$$776 + 699 + 748 + 783 > 3006.$$

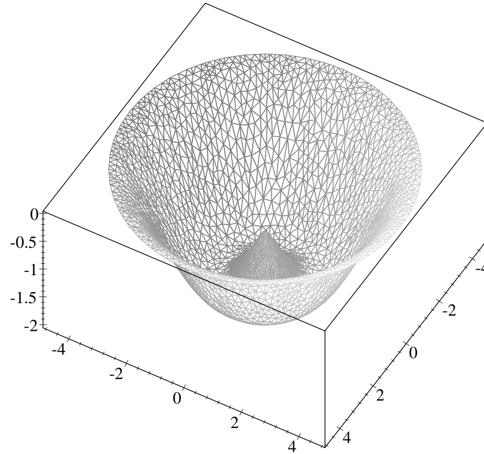


Computational results: All figures are for a run on Intel Paragon.

CG iterations	137
CG exit when	$\|r\|_{l^2} = 9.424367 \times 10^{-13}$ $(p, Ap)_{l^2} = 5.124849 \times 10^{-11}$. ($r$ = residual, p =search direction)
Times taken	For construction: 1.911063000094 secs For solving: 18.379607399926 secs
Errors	Relative l^2 error= 0.01492975182608 Relative l^∞ error=0.05385522079651

A plot of the computed solution is below:

Computed solution



4.2 A Neumann Problem

As of this writing, the code cannot handle all kinds of Neumann (or Robin) boundary conditions. Note that in our structures we do not have as yet a provision for incorporating “edges” of elements. This imposes a limitation in the kinds of Neumann boundary conditions we can input to the code. As it stands, a Neumann boundary condition of the form

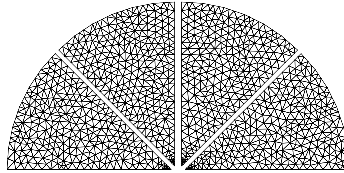
$$\frac{\partial u}{\partial n} = 0$$

on a segment of boundary can be prescribed as input (by assigning the appropriate attribute to the boundary vertices where the Neumann condition stands). Here $\hat{n}$ stands for the outward normal vector defined on the boundary.
P.D.E: $-\Delta u = \frac{sin(r)}{r} + cos(r)$.
Domain: Semi-circle about 0 of radius $3\pi/2$.
Boundary condition: $\frac{\partial u}{\partial n} = 0$, on straight edge $y = 0$, and $u = 0$ on the curved edge $r = 3\pi/2$.
True Solution: $cos(r)$.
Mesh: has 1026 vertices and 1917 triangles.
SubDomain splitting: The semi-circle is broken up into 4 equal angular sectors. The total number of triangles get split into 4 sub-domains as

$$476 + 471 + 488 + 482 = 1917$$

and the collection of vertices get divided as (some are shared)

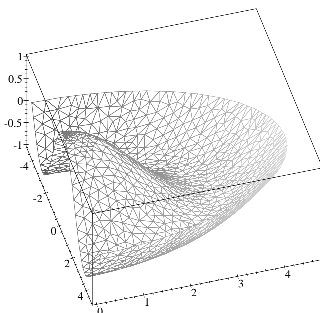
$$280 + 278 + 283 + 273 > 1026.$$



Computational results: All figures are for a run on Intel Paragon.

CG iterations	105
CG exit when	$\|r\|_{l^2} = 6.453330 \times 10^{-13}$ $(p, Ap)_{l^2} = 1.444293 \times 10^{-11}$. ($r$ = residual, p =search direction)
Times taken	For construction: 1.211754 secs For solving: 5.635507 secs
Errors	Relative l^2 error= 0.00935105546646 Relative l^∞ error=0.01059995013029

A plot of the computed solution follows:



4.3 Convergence with refinement

The unit square is broken into 5 sub-domains and triangulated. Though the runs were made in parallel, the parallel aspects of the code are not critical in this case. The aim here is to get finite element approximations to the true solution $x(1-x)y^2(1-y)$ of the problem

$$-\Delta u \;=\; 2y^2(1-y) - 2x(1-x)(1-y) + 4x(1-x)y$$

on successive refinements of a mesh, and verify the theoretically predicted error decay rate. The meshes are refined by breaking each triangle into 4 similar triangles (by creating 3 new edges joining the midpoints of each existing edge), so that the diameter of each triangle is halved on each refinement. The errors and other information are tabulated below (the first line is for the coarsest mesh).

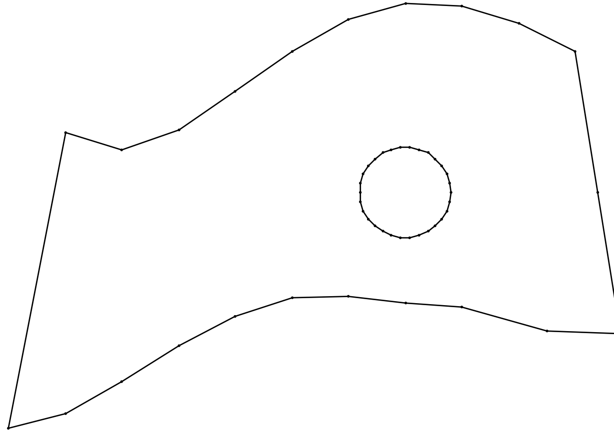
Diameter	Condition Number	No. of iterations	l^∞ error (relative)	l^2 error (relative)
h	2.799942e+01	27	0.01412106273429	0.00870073692159
h/2	1.216943e+02	57	0.00488643823240	0.00197861434598
h/4	5.584590e+02	121	0.00156859183164	0.00046988368503
h/8	2.489955e+03	200	0.00047208257276	0.00011696053813

Note that the condition numbers grow like $O(h^{-2})$ and the *relative* l^2 error (equivalent to error in L^2 norm) grows like $O(h^2)$, both as theoretically predicted.

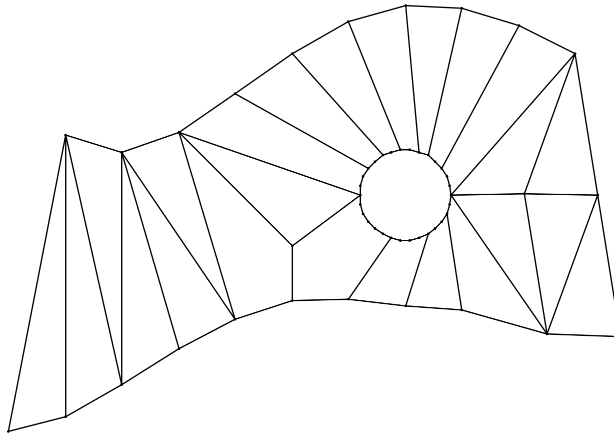
4.4 A problem on a general domain

The purpose of this example is to demonstrate the kinds of general domains on which problems can be solved through this code.

P.D.E: $-\Delta u(x,y) = 2y^2(1-y) - 2x(1-x)(1-y) + 4x(1-x)y.$
Domain: As shown below.



Boundary condition: Zero Dirichlet.
True Solution: Unknown.
Mesh: has 110,401 vertices, and 219,025 triangles. This is the largest size problem we have run.
SubDomain splitting: The domain is split into sub-domains as shown below.



Computational results: All figures are for a run on Intel Paragon.

CG iterations	910
CG exit when	$\lVert r \rVert_{l^2} = 9.797976 \times 10^{-13}$ $(p, Ap)_{l^2} = 1.631164 \times 10^{-11}$. ($r$ = residual, p =search direction)
Times taken	For construction: 26.442375 secs For solving: 1146.102972 secs

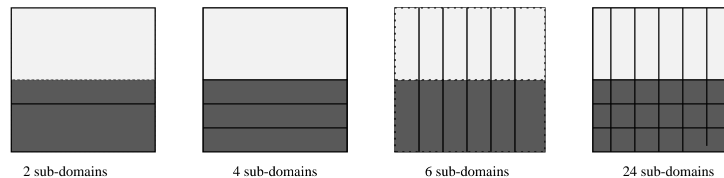
4.5 Parallel speedup

The two examples to be described now were run on both the Paragon and SGI. The maximum number of nodes accessible on Paragon was 24, and the maximum number of concurrent CPUs accessible on SGI was 8. Though the overall structure of the code run on both machines remain the same, they are not alike verbatim. The code run on Paragon uses MPI protocol for message passing. Parallelism on SGI is achieved through multiprocessing C++ compiler directives.

For this reason, on SGI, the speedup achieved has been far below those achieved on Paragon. Specifically, the problem arises because we found it necessary to switch between parallel and serial regions very often, the overhead for which is not negligible. Arguably this could be avoided, but at the cost of substantial code re-design, which we were not willing to do, especially as statement level parallelism using compiler directives, at least in its present state, cannot be ported across all shared memory machines. The only standard that we know offers portability across most shared memory machines now is the POSIX threads library. We have chosen not to use it because it involves some fairly low-level programming, and it did not seem worth the effort. Hopefully in the near future some standardisation effort for shared memory parallel machines will become successful enough to offer software that can take full advantage of the intrinsic parallel nature of our algorithm.

Example 1:

P.D.E: $-\Delta u = 2y^2(1-y) - 2x(1-x)(1-y) + 4x(1-x)y$.
Domain: Unit square $[0,1] \times [0,1]$.
Boundary condition: $u = 0$ on all boundary
True Solution: $x(1-x)y^2(1-y)$.
Mesh: has 18,233 vertices and 35,968 triangles. It is not shown here as some of the triangles are too small to be resolved. The lower half of the unit square is refined so that the triangles there are about 3 times smaller than those in the upper half.
SubDomain splittings: Several cases are run. First the mesh is split into 2 sub-domains and the problem is given to 2 concurrent CPUs. Next the same mesh is split into 4 sub-domains and given to 4 CPUs. Similar are the 6 sub-domain and 24 sub-domain cases. Note that the splittings are not done with minimising the interface lengths as an objective. In fact this issue is completely ignored for convenience. The sub-division is shown below (the darkly shaded areas represent the refined part).



Computational results: are tabulated below. There "Speedup" for k processors is calculated by

$$\frac{\text{Time taken to solve by 1 processor}}{\text{Time taken to solve by } k \text{ processors} \times k}$$

and the p and r at CG exit refers to the search direction and residual vectors respectively.

Parallel Nodes	Solving time (seconds)	Speedup (%)	No. of iterations	Relative Error in l^∞ / in l^2	CG exit $(p, Ap)_{l^2}$ / $\|r\|_{l^2}$
1	1034.51	–	346	0.00033000464806 0.00008488815349	1.521558×10^{-10} 9.004996×10^{-13}
2	535.24	96.6%	346	0.00032999040023 0.00008488818817	1.566095×10^{-10} 9.120379×10^{-13}
4	273.56	94.5%	346	0.00032999075854 0.00008488820869	1.471044×10^{-10} 8.922793×10^{-13}
6	181.17	95.2%	345	0.00032998559799 0.00008488792822	1.650998×10^{-10} 9.921209×10^{-13}
24	56.845	75.8%	345	0.00032998561433 0.00008488793009	1.644216×10^{-10} 9.898864×10^{-13}

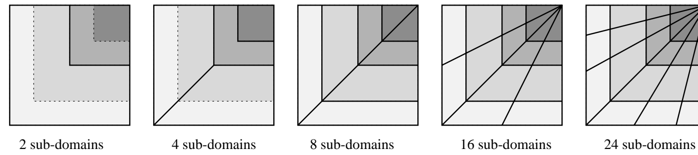
The results tabulated above are for a run on Intel Paragon. The speedup shown by 2, 4, and 6 node cases are good by any standards. It is to be expected that the 24 node case cannot exhibit the same level of speedup, because when the mesh is split into so many sub-domains the length of interface boundary increases. Also, the area of each sub-domain decreases. These two factors together make the ratio of the time spent in computation to the time spent in communication by each sub-domain lower.

Below is a similar table for a run on SGI Power Challenge for the same problem. As can be seen, parallel speedup is considerably lower than that achieved on the Paragon. It also drops sharply with increase in number of CPUs. We believe it is the overhead in spawning new threads at each switch to parallel region from a serial region, is the cause of this low speedup.

Parallel CPUs	Solving time (seconds)	Speedup (%)	No. of iterations	Relative Error in l^∞ / in l^2	CG exit $(p,Ap)_{l^2}$ / $\|r\|_{l^2}$
1	150.81	–	342	0.00033000436639 0.00008488814760	1.493181×10^{-10} 8.930991×10^{-13}
2	103.56	72.8%	346	0.00032999068207 0.00008488819512	1.488612×10^{-10} 8.951235×10^{-13}
4	72.25	52.2%	345	0.00032998561158 0.00008488793385	1.645427×10^{-10} 9.905752×10^{-13}
6	70.15	35.8%	346	0.00032999071632 0.00008488819685	1.480606×10^{-10} 8.936176×10^{-13}

Example 2:

P.D.E: $-\Delta u(x,y) = 2y^2(1-y) - 2x(1-x)(1-y) + 4x(1-x)y$.
Domain: Unit square $[0,1]\times[0,1]$.
Boundary condition: $u = 0$ on all boundary
True Solution: $x(1-x)y^2(1-y)$.
Mesh: has 53,162 vertices and 105,443 triangles. It is too refined to be shown here. The square sub-region $[0,0.5]\times[0,0.5]$ is more refined than other regions and the further sub-region $[0,0.25]\times[0,0.25]$ is highly refined.
SubDomain splittings: Several cases are run. First the mesh is split into 2 sub-domains and the problem is given to 2 concurrent CPUs. Next the same mesh is split into 4 sub-domains and given to 4 CPUs. Similar are the 6 sub-domain 8 sub-domain, 16 sub-domain and 24 sub-domain cases. As before, and as can be seen from the figures below the issue of minimising the interface created while splitting is completely ignored for convenience. The sub-domain decompositions are pictured below (note that the mesh triangles become more numerous as we approach the upper right corner).



Computational results: are tabulated below. There "Speedup" for k processors in the Intel Paragon is calculated by

$$\frac{\text{Time taken to solve by 2 processor}}{\text{Time taken to solve by } 2k \text{ processors} \times k}$$

while "Speedup" in the SGI Power Challenge run was calculated by

$$\frac{\text{Time taken to solve by 1 processor}}{\text{Time taken to solve by } k \text{ processors} \times k}$$

This difference was necessitated by the fact that no Intel Paragon node had enough memory to run the whole problem by itself, so that no data for the single node case could be collected there. The p and r at CG exit refers to the search direction and residual vectors respectively.

Parallel Nodes	Solving time (seconds)	Speedup (%)	No. of iterations	Relative Error in l^∞ / in l^2	CG exit $(p, Ap)_{l^2}$ / $\|r\|_{l^2}$
2	3531.45	–	799	0.00012200800933	2.798025×10^{-10}
				0.00003043137783	9.804515×10^{-13}
4	1893.84	93.2 %	799	0.00012200871414	2.364306×10^{-10}
				0.00003043138931	9.799664×10^{-13}
8	952.372	92.7%	797	0.00012200897346	2.533462×10^{-10}
				0.00003043141280	9.960237×10^{-13}
16	506.734	87.1%	796	0.00012200823221	2.368166×10^{-10}
				0.00003043138413	9.452165×10^{-13}
24	348.577	84.4%	796	0.00012200786428	3.116723×10^{-10}
				0.00003043137710	9.830069×10^{-13}

The results tabulated above are for a run on Intel Paragon. Below is a similar table for a run on SGI Power Challenge of the same problem.

Parallel CPUs	Solving time (seconds)	Speedup (%)	No. of iterations	Relative Error in l^∞ / in l^2	CG exit $(p, Ap)_{l^2}$ / $\|r\|_{l^2}$
1	1132.12	–	803	0.00012200353820	3.385142×10^{-10}
				0.00003043133550	9.503178×10^{-13}
2	720.79	78.5%	802	0.00012200679749	3.264349×10^{-10}
				0.00003043136554	9.356017×10^{-13}
4	428.45	66.1%	800	0.00012200708385	2.848791×10^{-10}
				0.00003043136831	9.151883×10^{-13}
8	327.47	43.2%	797	0.00012200885430	2.874384×10^{-10}
				0.00003043139817	9.954389×10^{-13}

Acknowledgments

This work has been supported by EPA grant # R 825207-01-1.

References

1. Petter E. Bjorstad, William Gropp and Barry Smith. *Domain decomposition : parallel multilevel methods for elliptic partial differential equations.* Cambridge University Press, 1996.

2. James H. Bramble, J. E. Pacsiak, A. H. Schatz. *The construction of preconditioners for elliptic problems by substructuring. I.* Math. Comp. v.47, 1986, pp 103-145.

3. William Gropp, Ewing Lusk and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994.

4. Claes Johnson. *Numerical solution of partial differential equations by the finite element method.* Cambridge University Press, 1995.

5. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra. *MPI: The Complete Reference.* MIT Press, 1996.

6. Bjarne Stroustrup. *The C++ Programming language.* Addison-Wesley Publishing Company, 1995.