Scientific Computing Tool

Ajaykumar Poondla
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
Email: ajaykumarpoondla@hotmail.com

Haining Liu
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
Email: hliu9834@hotmail.com

Jeff.S.McGough
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
Email: jeff.mcgough@sdsmt.edu

ABSTRACT

The success of high performance computing in modeling scientific and engineering applications motivates the development of ambitious applications. An efficient solution for solving scientific problems on a cluster has been and is still of high interest. In this paper, we present the Scientific Computing Tool.

The Scientific Computing Tool provides a Graphical User Interface (GUI) in which the user can enter geometric shapes in one, two or three dimensions, and information about the shapes. The tool will then take the necessary parameters for solving the linear elliptic partial differential equations on a rectilinear domain. The program will generate a sequential code or parallel code as requested by the user for a finite difference approximation defined on the domain using the iterative methods Jacobi, Gauss Seidel, Successive Over Relaxation methods. The serial code is implemented in C. The parallel code is implemented in C++ using POOMA library. This tool allows the user to compile, run and generate a graph in the GUI using Gnuplot.

Keywords: Scientific Computing, Domain Decomposition, Elliptic PDE, POOMA, Jacobi, Gauss and SOR, Graphical User Interface.

Introduction

The computer science department at SD School of Mines and Technology started a Beowulf Cluster Project (Cluster Computing and Visualization) in spring 2002. This project aims at using "off-the-shelf" components to build a parallel computer to perform the numerous complex operations required for visualization of large data sets. We decided to come up with a tool that simplifies the code development in Scientific Computing. The purpose of Scientific Computing Tool is to create a Graphical User Interface in which you can enter in geometric shapes in one, two or three dimensions. The tool then will take the necessary parameters for solving a linear elliptic partial differential equation on a rectilinear domain and generate code for a finite difference approximation defined on the domain using the stationary iterative methods Jacobi, Gauss Seidel and Successive Over Relaxation methods. Since domain decomposition methods are based on partitioning of the domain of the physical problem and each sub domain can be handled independently, this tool is very effective on a cluster.

This tool streamlines the development of scientific codes. This includes the process of modeling, discretization, solving and parallelization of problems from various fields of applications[5]. The various fields of Scientific Computing include weather prediction, Seismic data processing, astrophysics, Nuclear Engineering and Image processing. This tool is readily accessible to scientific application developers whose background does not include computer science. This tool leverages existing sophisticated codes like POOMA to achieve this. The Parallel Object Oriented Methods and Applications (POOMA) Toolkit is an open-source software for writing high performance Scientific Computing Programs on parallel computers, which was originally developed by scientists at Advanced Computing Laboratory at Los Alamos National Laboratory (LANL), and is maintained by CodeSourcery, LLC (http://www.codesourcery.com) [8,9]. Also, a Cheetah messaging library provided by LANL, an underlying messaging library Messaging Passing Interface (MPI) from Argonne National Laboratory Computation Institute, as well as MM Shared Memory Library are coupled with POOMA, in order to take the advantages of high performance parallel computing of POOMA over multiple processors or cluster system

The scientific computing tool provides an easy-to-use interface and is a far better means of communication than text-based alternatives. Extensive use of visual navigation features such as buttons, menus and trees and intuitive manipulation of data will make this tool convenient for the scientific computing world. The tools needed to build and test Java programs are available without charge. Sun makes the Java Development Kit (JDK) available over the Internet (at http://www.javasoft.com/), where any individual can download it. The JDK--which includes the Java compiler and interpreter, among other tools is undoubtedly very simple to use.

Description of the problem

Partial Differential Equations describe the modeling of physical processes taking place in our surroundings. A Partial Differential Equation (PDE) is a type of equation in which the unknown can represent some of the things like the temperature, or the shape of the wave, or stress in a bent piece of metal. A PDE is an equation involving one or more partial derivatives of an unknown function of multiple variables [3].

A general second-order partial differential equation looks like this.

$$a\frac{\partial^2 u}{\partial x^2}+b\frac{\partial^2 u}{\partial x\partial y}+c\frac{\partial^2 u}{\partial y^2}+d\frac{\partial u}{\partial x}+e\frac{\partial u}{\partial y}+fu+g=0$$

where *a, b, c, d, e, f* and *g* can be functions of both the independent variables *x* and *y* and the dependent variable *u*. This equation is said to be *elliptic* when $b^2-4ac<0$, *parabolic* when $b^2-4ac=0$ and *hyperbolic* when $b^2-4ac>0$. Elliptic partial differential equations arise usually from equilibrium or steady-state problems and their solutions [3].

The goal is to solve elliptic partial differential equation

$$a\frac{\partial^2 u}{\partial x^2}+b\frac{\partial^2 u}{\partial x\partial y}+c\frac{\partial^2 u}{\partial y^2}+d\frac{\partial u}{\partial x}+e\frac{\partial u}{\partial y}+fu+g=0$$

in the domain $\Omega=(x_0,x_1)x(y_0,y_1)$, subject to the Dirchlet boundary condition $u=G_i$ on $\partial\Omega$. This problem can be solved using the method of finite differences. The method of finite differences is easy to implement and it gives good results of boundary value problems. In this method, the derivatives appearing in the equation and the boundary conditions are replaced by their finite difference approximations [2]. Then the given equation is changed to a difference equation, which is solved by iterative procedures. By replacing the given derivatives with their finite difference approximations we get

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_{(i,j)}\cong\frac{u(i+1,j)-2u(i,j)+u(i-1,j)}{h^2}\quad\left(\frac{\partial^2 u}{\partial y^2}\right)_{(i,j)}\cong\frac{u(i,j+1)-2u(i,j)+u(i,j-1)}{k^2}$$

$$\left(\frac{\partial u}{\partial x}\right)_{(i,j)}\cong\frac{u(i+1,j)-u(i-1,j)}{2h}\quad\left(\frac{\partial u}{\partial y}\right)_{(i,j)}\cong\frac{u(i,j+1)-u(i,j-1)}{2k}$$

$$\left(\frac{\partial^2 u}{\partial x\partial y}\right)_{(i,j)}\cong\frac{u(i+1,j+1)+u(i-1,j-1)-(u(i-1,j+1)+u(i+1,j-1)}{4hk}$$

where h is the step size between discrete points and is equal to $\frac{(b-a)}{N}$ and k is the discrete points and is equal to $\frac{(d-c)}{M}$.

The above equation can be solved by dividing Ω into *NxM* cells and applying central difference formula at each of *(N – 1) x (M – 1)* interior grid points.

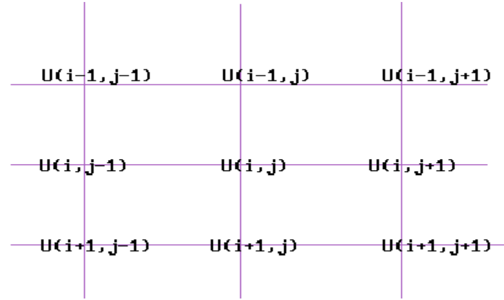


Figure 1

For the grid point in the ith row and j^{th} column of the grid gives the formula

$$A_1u(i+1,j)+A_2u(i-1,j)+A_3u(i,j+1)+A_4u(i,j-1)+$$
$$A_5u(i+1,j+1)+A_6u(i+1,j-1)+A_7u(i-1,j+1)+A_8u(i-1,j-1)+A_9u(i,j)+g4h^2k^2=0$$

Equation 1

$$u(i,j)=\frac{-1}{A_9}\left\{\begin{array}{l}A_1u(i+1,j)+A_2u(i-1,j)+A_3u(i,j+1)+A_4u(i,j-1)+\\ A_5u(i+1,j+1)+A_6u(i+1,j-1)+A_7u(i-1,j+1)+A_8u(i-1,j-1)+g4h^2k^2\end{array}\right\}$$

where $1\le i\le Nx,\quad 1\le j\le Ny,$

$A_1 = 4ak^2+2dhk^2,\quad A_2 = 4ak^2-2dhk^2,\quad A_3 = 4ch^2+2ehk^2,$

$A_4 = 4ch^2-2ehk^2,\quad A_5 = bhk,\quad A_6 = -bhk,\quad A_7 = -bhk,$

$A_8 = bhk,\quad A_9 = 4fh^2k^2-8ch^2-8ak^2$

Equation1 can be solved using stationary methods Jacobi, Gauss-Seidel, Successive Over Relaxation and non-stationary methods like Conjugate Gradient Method and GMRES.

Existing Approach

Grid based problems are common in scientific computing. An efficient solution to solve the partial differential equations has been and is still of interest. The following are the tools that are commonly used by scientific computing world.

Matlab: Matlab is a high level interpreted programming language generally used for high performance numerical computation and visualization The Matlab PDE solver, *pdepe*, solves initial-boundary value problems for systems of parabolic and elliptic PDEs in the one space variable *x* and time *t*.

Mathematica: Mathematica[9], software developed by the company Wolfram Research, is a numeric and symbolic calculation system that incorporates an excellent programming language and the capacity of integrating calculations, graphics and text, in oneself document electronic, called notebook.

Maple: The Maple system is an advanced, mathematical problem solving and programming environment.

PETSc: PETSc is a software library that provides data structures and routines for the solving scientific applications modeled by partial differential equations on parallel or serial computers. To provide portability across networks of workstations PETSC uses customized message-passing system.

POOMA: POOMA (Parallel Object-Oriented Methods and Applications) is a collection of templated C++ classes for writing parallel PDE solvers using finite-difference and particle methods. It provides a variety of tools in supporting scientific computing with features of [6],

- Containers and other abstractions for scientific computation,
- Support for a variety of computation modes, such as data parallel, stencil-based computations, and lazy evaluation, as well as parallel and distributed computation programs writing.
- Robots of all inter process communication for parallel and distributed routines, out of order execution, and loop rearrangement for efficient program execution.

With POOMA high-level abstractions, the programs written in POOMA are much shorter than conventional FORTRAN or C programs. The code is also easier to debug. The toolkit is compatible with most of C++ compliers.

Limitations in existing software

Solving scientific applications generally involves a great deal of arithmetic. One of the major problems faced by the programmer is to convert the mathematical concept into efficient algorithm developed in some programming language. Though the existing software has come up with a programming environment, still the user is expected to write his code for a specified input. This becomes an uphill task to write a parallel code, which requires a different algorithm.

Scientific Computing Tool

This tool generates code a serial code and parallel code for a user specified input to solve a general elliptic partial differential equations. The serial code is implemented in C. The parallel code is implemented in C++ using POOMA library.

The user interface provides a similar look and feel to a GUI based program and is implemented using Java Swing for increased portability to other operating systems. To the greatest degree possible, the layout and functionality of menus, dialog boxes, and toolbars follows the standard GUI guidelines for Software Design.

The following are the important classes used in our system for generating the code to solve elliptic partial differential equations.

Start: This is the starting point of the tool. This brings up the splash screen and provides the user the details of the system he is running and it directs to the pdetool.

Pdetool: This is the heart of the tool. This class represents the view of the hierarchical data of the PDE.

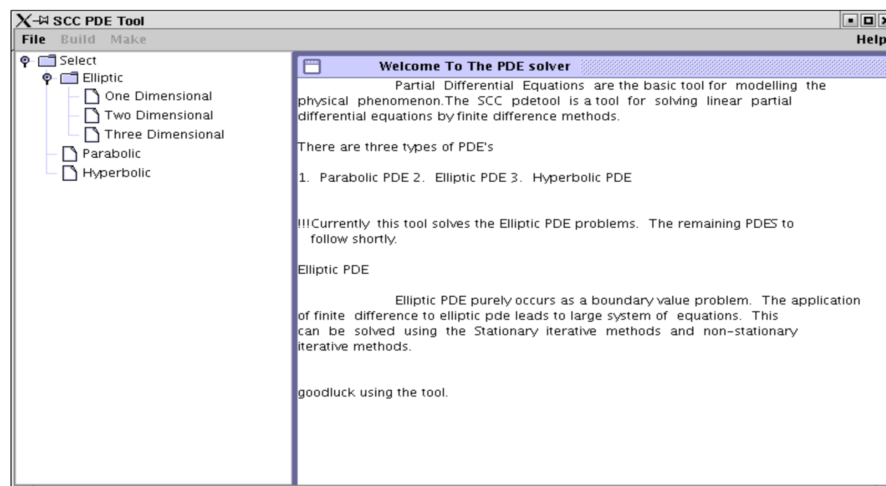


Figure 2

As the preceding figure shows, the nodes of the tree contain Elliptic, Parabolic, Hyperbolic, and elliptic node containing the children one dimensional, two dimensional, three-dimensional. When the user right clicks one dimensional, two dimensional, three-dimensional the corresponding sub nodes containing line, rectangle, cube, are obtained. When the user clicks the node line, rectangle, ellipse, circle and cube the corresponding object is created in an internal frame. The internal frame can be resized and moved through out the window. When the user clicks line/rectangle/cube, a wizard is created that takes in the necessary inputs for handling the PDE problem. This is handled in the host class.

Host: This contains a handle to an array of panels each representing one-step wizard process. This also captures the input given by the user.

Frame: Based on the input entered by the user this class writes them in a syntactically correct format and calls the corresponding solver to solve the particular problem.

The following is the class diagram of the scientific computing tool.

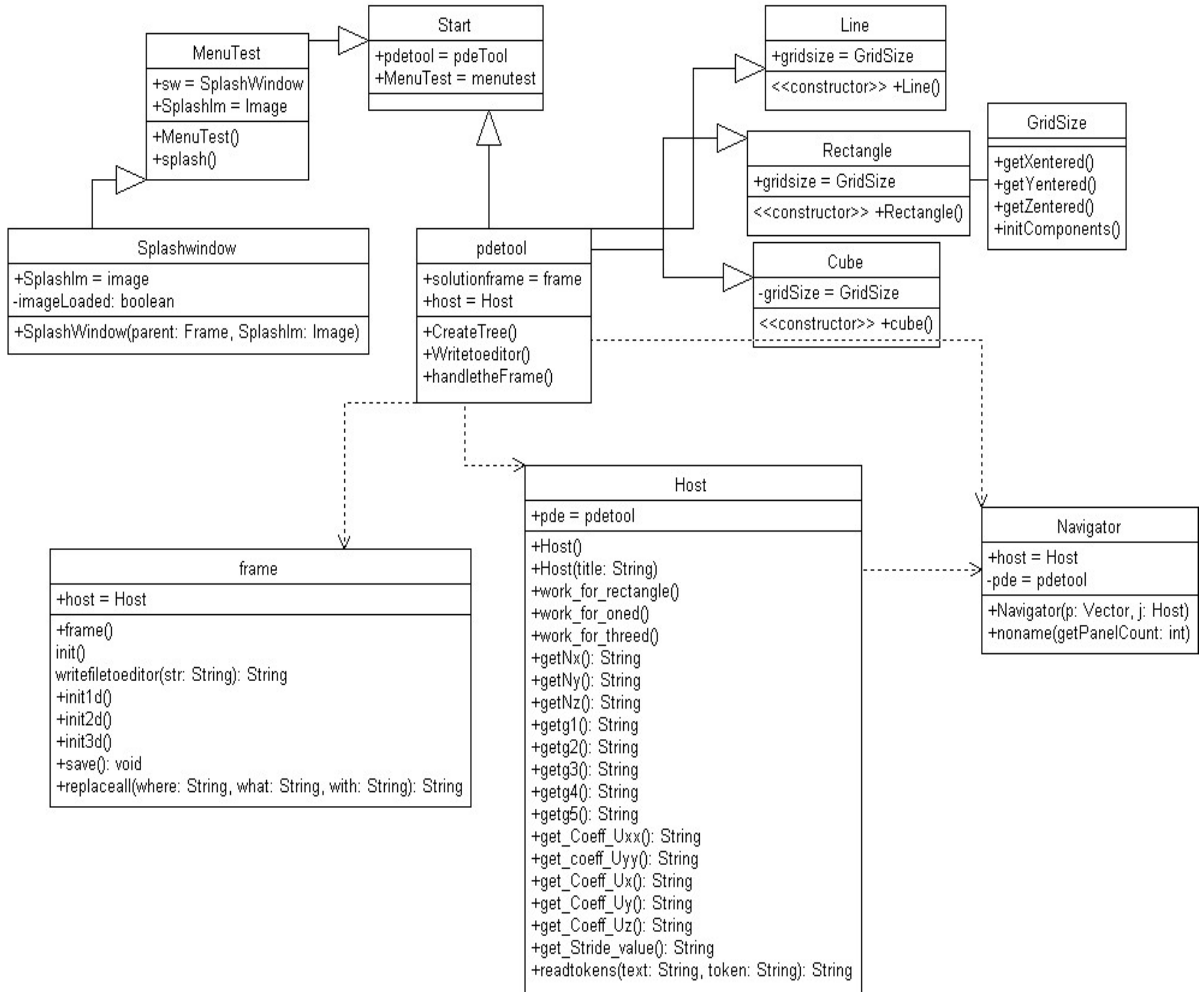


Figure 3

A parallel sample code generated by use of scientific computing tool for solving a variable Poisson equation with variable coefficients and boundary conditions is shown below.

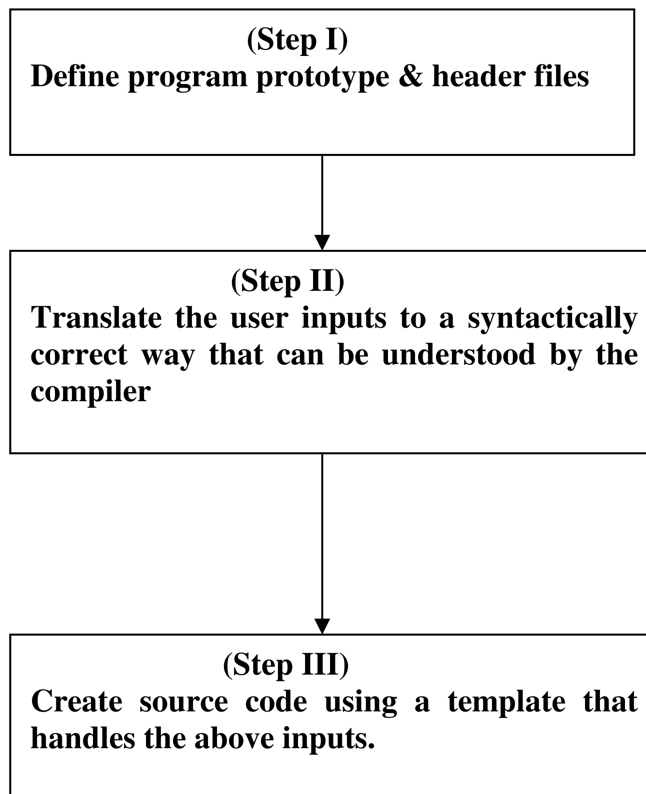
```
#include "Pooma/Arrays.h"
#include <iostream>
#include <fstream.h>

#define Nx 10
#define Ny 10
#define stride 2
#define Iteration 200

#define hsquare ((double)Nx-1.0)/((double)(Nx*Nx))
#define ksquare ((double)Ny-1.0)/((double)(Ny*Ny))

//Define the coefficients a, b, and c.
inline double A(double, double);
inline double B(double, double);
inline double C(double, double);

inline double A(double x, double y) {return x+y;}
inline double B(double x, double y) {return x+y;}
inline double C(double x, double y) {return x+y;}

//Boundary conditions to be applied.
inline double g1(double,double);
inline double g2(double,double);
inline double g3(double,double);
inline double g4(double,double);

inline double g1(double x,double y){ return x+y;}
inline double g2(double x,double y){ return x+y;}
inline double g3(double x,double y){ return x+y;}
inline double g4(double x,double y){ return x+y;}

// Apply a Jacobi iteration on the given domain.
void ApplyJacobi(
   const Array<2>  & V,  // the domain to be solved.
   const Array<2>  & a,  // the domain to be solved.
   const Array<2>  & b,  // the domain to be solved.
   const Array<2>  & c,  // the domain to be solved.
   const Range<1>  & I,  // x axis subscript
   const Range<1>  & J   // y axis subscript
)
{
  V(I,J) = (1.0/(2.0*(ksquare*a(I,J) + hsquare*b(I,J)))
      )*
      ( (ksquare*a(I,J))*(V(I+1,J) + V(I-1,J))
   +    (hsquare*b(I,J))*(V(I,J+1) + V(I,J-1))
   -    c(I,J)*hsquare*ksquare
      );
}

//Calculate the sum of squares errors in a 2D Array.
//No modification needs below.
template<class ValueType, class EngineTag>
ValueType sum_sqr (const Array<2, ValueType, EngineTag>
&A)
{
  ValueType sum = 0.0;

  int begin_0 = A.first(0), end_0  = A.last(0), begin_1 =
A.first(1),
     end_1  = A.last(1);

  // Must block before scalar loop.
  Pooma::blockAndEvaluate();
```

```
int
main( int argc,      // argument count
      char *argv[]   // argument list
)
{
  ofstream osf;

  // Initialize Pooma.
  Pooma::initialize(argc, argv);

  // The array to be solved, zero out.

  Array<2> V(Nx, Ny);
  V = 0.0;

  // The right hand side function of the equation f(x,y):
  //Initialization
  Array<2> a(Nx, Ny);
  a = 0.0;

  Array<2> b(Nx, Ny);
  b = 0.0;

  Array<2> c(Nx, Ny);
  c = 0.0;

  // Must block before scalar code.
  Pooma::blockAndEvaluate();

  //Define f(x,y)
  for(int x = 0; x < Nx; ++x)
            for(int y = 0; y < Ny; ++y){
    a(x,y) = A(1,0); //coefficient of uxx
    b(x,y) = B(1,0); //coefficient of uyy
    c(x,y) = C(0,0); //right hand side function
  }

  c(Nx/2,Ny/2) = (double) -Nx/2.0;

  for(int x = 0; x < Nx; ++x) {
    V(x,0)    = g1(0,0);
    V(x,Ny-1) = g2(0,0);
  }

  for(int y = 0; y < Ny; ++y){
    V(0,y)    = g3(0,0);
    V(Nx-1,y) = g4(0,0);
  }

  // The interior domain, now with number of stride.
  // No modification is needed below at this time.
  Range<1> I(1, Nx-(stride+1), stride), J(1, Ny-(stride+1), stride);

  // Iterate till converged, or maximum Iteration steps.
  double SSErr = 0.01; // anything greater than threshold
  int iteration;
  for (iteration=0; iteration < Iteration && SSErr > 1e-6;
++iteration)
  {
    //Red block
    ApplyJacobi (V, a, b, c, I,   J);
    ApplyJacobi (V, a, b, c, I+1, J+1);
    //Black block
```

```
 for (int x = begin_0;  x <= end_0; ++x)
 {


   for (int y = begin_1; y <= end_1; ++y)
   {
     ValueType value = A.read(x, y);
     sum += value * value;
   }
 }

 return sum;
}
```

```
   ApplyJacobi (V,a, b, c, I+1, J);
   ApplyJacobi (V,a, b, c, I,   J+1);
   //Compute residual.
   SSErr = sum_sqr ( (V(I+1,J) + V(I-1,J))*ksquare*a(I,J)
        +  (V(I,J+1) + V(I,J-1))*hsquare*b(I,J)
        -  (c(I,J)*hsquare*ksquare
        +  2.0*(ksquare*a(I,J) + hsquare*b(I,J))*V(I,J)
          ));
  }

  // Print out the result.
  std::cout << "Iterations = " << iteration << std::endl;
  std::cout << "Residual = "   << SSErr  << std::endl;
  std::cout << "Solved domain size in " << Nx << " x " << Ny <<
"." << std::endl;
  //Write the result to file.
  osf.open ("result.txt");
  osf << V << std::endl;
  osf.close ();

  // Clean up and report success.
  Pooma::finalize();
  return 0;
}
```

The entire code generation was divided into three steps. The flow diagram shown below describes how the code generation was handled effectively within the scientific computing tool.



Results and Discussion

The goal of this project is to come up with a small prototype in building a generic tool that is suitable for scientific computing on a cluster. As the first step, we tested the tool for the linear elliptic partial differential equations. The generated code was compiled and run on the cluster. In the case of parallel code execution, a cluster with 4 nodes, running Pentium III 440 MHz processor and 256 MB RAM was used for testing performance. 10/100 Base-T Faster Ethernet card is used for communications between nodes.

Figure 4 illustrates the performance of the three solvers of using different algorithms for solving a Poisson equation on a rectangular grid with sequential code implemented in C.

A simple method for solving the above equation is to iteratively apply the equation until there is little change in u. This is the ***Jacobi*** iterative method for solving the equation. A modification of this process, called the ***Gauss-Seidel*** method, may find the solution in fewer iterations. As described in *Introduction to Parallel Algorithms and Architectures*, page 98, this method suggests that we can calculate the new values of all the even-parity points using equation 1; then calculate the new values for all of the odd-parity points using the values just calculated for the even parity points[1]. The *parity* of the point (i/N , j/N) is defined to be even if $i + j$ is even and odd if $i + j$ is odd. This will cause the method to converge in fewer iterations, but may not be beneficial for small matrices since this requires two communications per iteration instead of one for Jacobi's method. Gauss-Seidel iteration method can be further be improved by changing the convergence rate using the ***Successive over relaxation (SOR)*** method. Application of SOR to the above set of equation gives

$u_{i,j}^{m+1} = (1-\omega)u_{i,j}^{m} + \omega u_{i,j}^{m+1}$ Where $1 < \omega < 2$.

The value of ω is normally $\dfrac{2.0}{\left(1+\sin\left(\dfrac{2\pi}{N_x + N_y}\right)\right)}$.

Where N_x, N_y are the size of the grid along the x and y-axis.

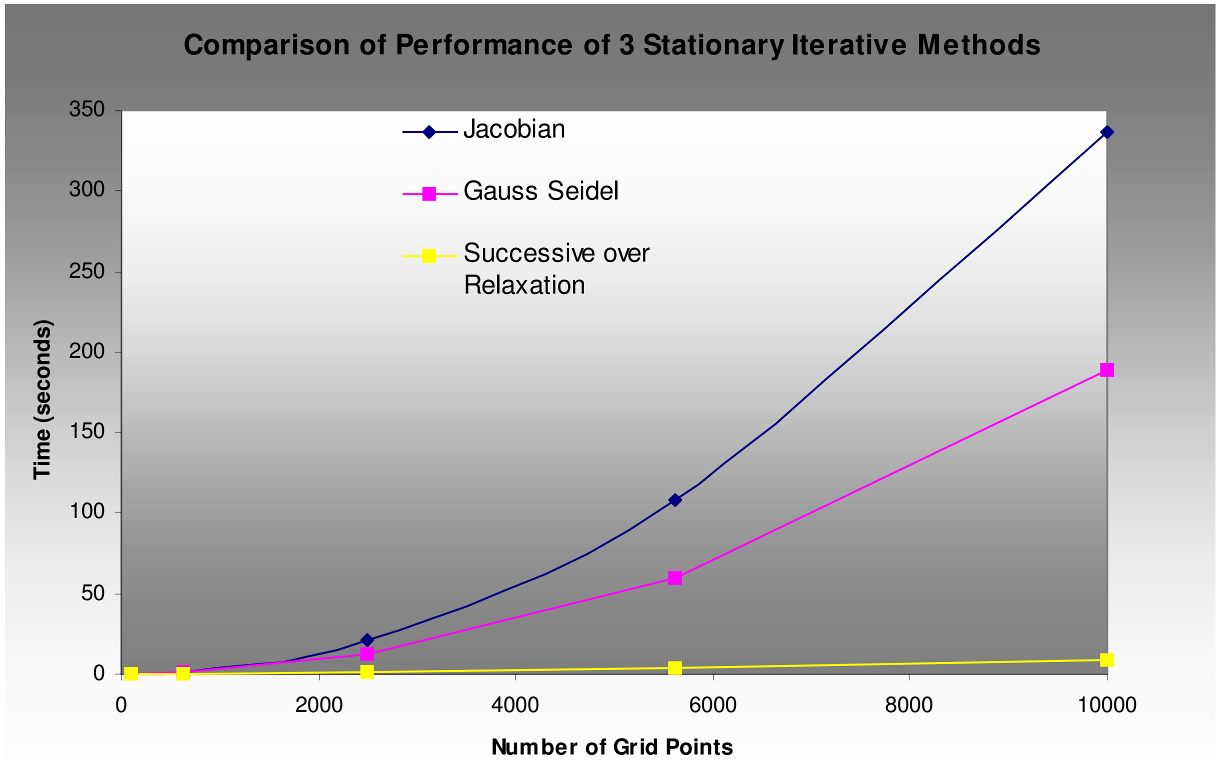
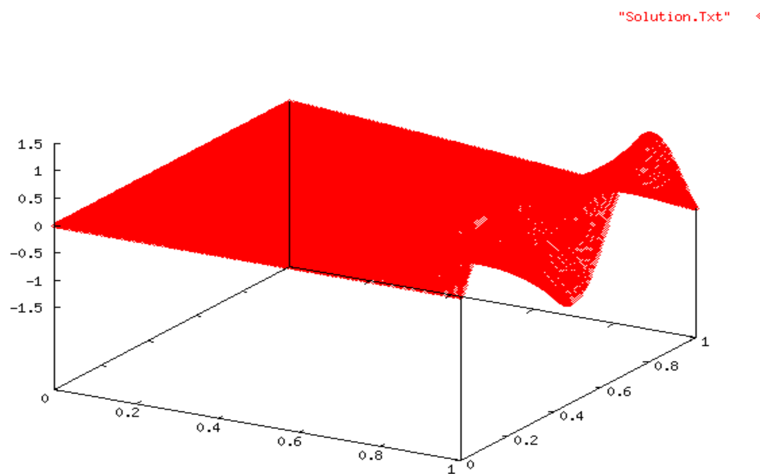


Figure 4. Comparison of performance of three algorithms implemented in the serial code.

The following is a graph generated by the tool for solving $\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial x \partial y} + \frac{\partial^2 u}{\partial y^2} = 0$ in the domain *Ω=(0,1)x(0,1),* subject to the dirchlet boundary condition *u = sinh(x)*sin(10*y)* on *∂Ω*..



As a comparison, Figure 5 shows the performance of execution of a parallel code by applying Jacobi Iteration Relaxation method to solve the same problem over 4 nodes cluster system. One may wonder how this parallel code could be so efficient, it might have been too good to be true. However, with careful examination of the sample parallel code, one can observe that the implementation of Jacobi Iteration Relaxation algorithm has taken the benefits of data parallel expression on POOMA. As it has been show in the sample code, both parameters and variable (unknown) were defined as Array objects. The initialization of these array objects was done with single statement, but none for loop presents. The Range objects shown in the sample code (one dimension in this case), benefit the representation of index sequences with non-unit strides, which provides an efficient way to define non-adjacent array elements in parallel fashion. The Red/Black block operations within iteration loop further improve the parallel performance on computing, it reduces the amount of memory that a program requires [9].

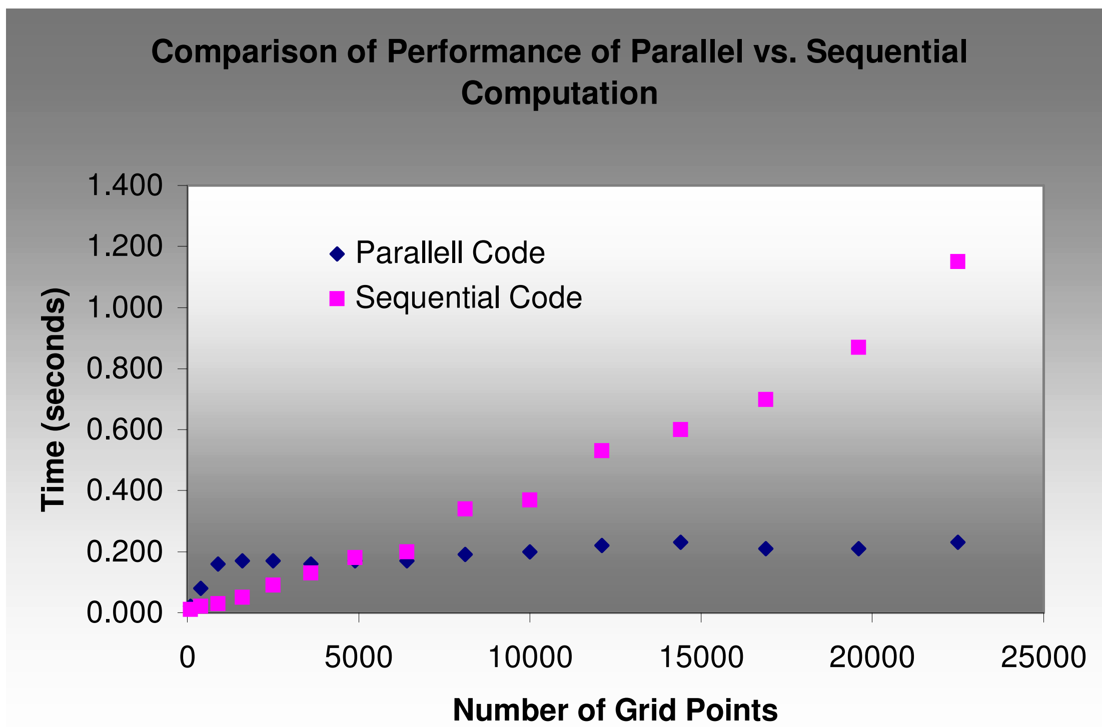


Figure 5. Comparison of performance of parallel versus sequential code.

Another important point should be addressed here regarding POOMA highly abstractive objects is that it uses a type of object called Engine, a key to the POOMA high performance [7]. An engine is an abstractive object implemented in POOMA, which performs the low-level value storage, computation, and element wise access for a container [6]. The function sum_sqr () shown in the sample code takes the advantages of Engines. It is completely different form other languages that use whole-array operations, which usually requires temporary array for holding data. In fact, during the computing process, the arrays do not store data in POOMA, but act as handles on an engine. The engine knows how to evaluate and return values on the given sets of indices

corresponding the predefined arrays. That is, engine can reference data storage directly, and translate a set of indices into a value by looking up the value based on the indices in memory.

An interesting observation from Figure 5 is the overhead of interprocess communication among the nodes of cluster for parallel code execution. Initially, the grid size is relatively small, such that sequential code execution is much faster than parallel code, because the communication overhead can't be eliminated for the parallel code. In fact, for a small grid size, the parallel code is not the first choice for efficiency concern.

Conclusions & Future Enhancements

In this project, it is aim to develop a tool that simplifies the code development in scientific computing. The tool can generate both serial and parallel code for a general elliptic PDE problem according to the user specified input in a fashion of less time consuming with user-friendly interface. With some simple modifications (for example, adding a node to the tree structure), it can be extended to solve different kind of problems. Current work that we have done is attempting to solve linear elliptic partial differential equations using the stationary methods. Currently we are working on the non-stationary methods such as Conjugate Gradient method and GMRES and they will be added in the next release. This tool solves elliptic partial differential equations on a rectilinear domain. This can be extended to linear two-dimensional first order systems of elliptic partial-differential equations (PDE's) and associated boundary conditions over a finite union of rectangles. Details can be found in [8]. Scientific Computing Tool currently can generate parallel code that uses data parallel mechanism. With the cluster system, it would be more efficient that machine parallelism could be introduced with the consideration of load balancing over all nodes of whole cluster system. In order to have full beneficiary of POOMA, the tool should be able to support parallel computation over a fully distributed cluster system.

References

1. Thomson, L.F. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees and Hypercubes.* San Mateo, California: Morgan Kaufmann, 1992.
2. Grewal, B.S. *Higher Engineering Mathematics*, Khanna Publications, 19xx.
3. Duchateau, P., and ZachMann, D.W., *Partial Differential Equations Schaum's outline series in mathematics,* Publisher, 19xx.
4. Naughton, P. and Schildt, H., *The Complete Reference Java* 3rd. edition, Tata McGraw-Hill, 19xx.
5. Norton, C.D., Thesis work on *Object oriented paradigms in scientific computing*, Department of Computer Science Rensselaer Polytechnic Institute, Troy, New York, 12180-3590, USA.
6. Oldham, J.D., *POOMA A C++ Toolkit for High Performance Parallel Scientific Computing,* Codesourcery, LLC. March 1st. 2001.

7. Williams, T.J., Reynders, J.W., Humphrey, W.F., and Cummings, J.C., *POOMA User Guide, Parallel Object-Oriented Methods and Applications,* Los Alamos National Laboratory, 1999.
8. HOHN.M. *On the Solution of Mixed Boundary Value Problems in Elasticity*. Ph.D. thesis, Department of Mathematics, University of Utah, SaltLake City, UT, USA,Dec.2001.
9. *Educational applications of Mathematica (*http://www.xtec.es/~fgomez/apmath-e.html)

Acknowledgements

We would like to thank Dr.Gregg Stubbendieck for the valuable suggestions in designing the interface. We would like to thank the Beowulf Cluster group at SDSM&T for their support. We would like to thank all the individuals who are directly or indirectly involved in the successful completion of this project.