ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Advanced Parallel Computing for Scientific Applications
Autumn Term 2010

Prof. I. F. Sbalzarini
ETH Zentrum, CAB E64.1
CH-8092 Zürich

Prof. P. Arbenz
ETH Zentrum, CAB H89
CH-8092 Zürich

Exercise 10

Release: 30 Dec 2010
Due: 14 Dec 2010

Red Black Gauss Seidel Method

The *Red Black Gauss Seidel* method is an iterative method used to solve a linear system of equations resulting from the *finite difference* discretization of partial differential equations. The Red Black Gauss Seidel method can be considered as a compromise between the *Jacobi* and the *Gauss Seidel* iterative scheme. In this exercise, we will solve the simple partial differential equation :

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 1.0 \tag{1}$$

on the unit square domain, the two dimensional co-ordinates x and y, and the Dirichlet boundary conditions

$$u(x, y) = 0 \qquad x, y \in \partial]0, 1[^2 \tag{2}$$

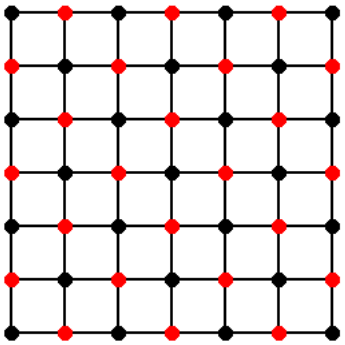

Figure 1: Red black ordering of nodes on Cartesian grid

The finite difference discretization of eq.1 on a Cartesian grid results in the following system of equations

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h_x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{h_y^2} = 1 \qquad i \in \{1, \ldots, N_x - 1\}, j \in \{1, \ldots N_y - 1\}$$

where h_x and h_y are the node spacing in x and y directions. Assuming $h_x = h_y = h$, each nodal value $u_{i,j}$ can be calculated as :

$$u_{i,j} = \frac{u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - h^2}{4} \tag{3}$$

As seen from Eq.3 and Fig.1, $u_{i,j}$ at a black node can be calculated using the nodal values at the 4 adjacent nodes which are all red. Thus, in the first pass, $u_{i,j}$ will be calculate in parallel at all the red nodes. In the second pass, $u_{i,j}$ will be calculated in parallel at the black nodes. This procedure will be repeated until the residual is reduced below certain tolerance limit while maintaining the specified boundary conditions.

Domain decomposition and communication between processors

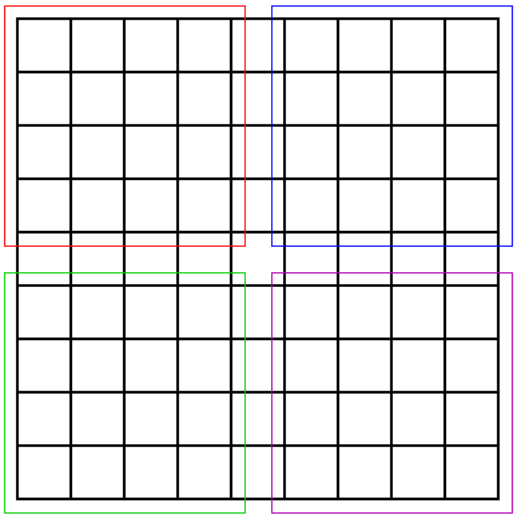

Figure 2: Domain decomposition

As indicated in the Fig.2, the square domain is broken down into smaller blocks and each block is assigned to a processor.

- Each processor will allocate memory for all the internal nodes as well as for nodes lying on `ghost layer` on each side.
- The ghost layers will be populated by values from adjacent processors.
- Each processor will calculate $u_{i,j}$ only at the internal nodes.

Question 1: Parallel Implementation of Red Black Gauss Seidel Method

Implement the following functions in the file `red_black.cpp`

- `boundary_conditions(...)` : Apply the boundary conditions as specified in Eq.2.
- `initialize(...)` : All iterative methods begin with an initial guess for the unknown values. Initialize all the internal nodal values $u_{i,j}$ with 1.0
- `communicate(...)` : Fill in the ghost layer with appropriate values from adjacent processors.
- `red_black_GS(...)` : Implement the *Red Black Gauss Seidel* method as described in the previous sections.

File `red_black.c` has the sequential version of the *Red Black Gauss Seidel* method. Compare the results of your implementation and comment on the speedup.

Note :

1. Instead of using the provided source code you can write your own program using `MPI_Cart_create(...)`.
2. If possible, try to achieve latency hiding by using non blocking communication.

Question 2: Preconditioned Conjugate Gradient using Trilinos

The *Conjugate Gradient* method is an algorithm for the numerical solution of particular systems of linear equations, namely those whose matrix is symmetric and positive-definite. In `Trilinos`, `AztecOO` package provides an interface for several linear solvers including *Conjugate Gradient*. The source code in `cg.cpp` uses an AztecOO object to solve a linear system of equations using *Preconditioned Conjugate Gradient*.

a) Read the `AztecOO`[1] documentation in order to understand the source code provided in the file `cg.cpp`

b) **Optional** Write your own implementation of the *preconditioned CG method* (`pcg`) using the Trilinos framework. The PCG-algorithm reads as follows:

Choose $\mathbf{x}_0$. Set $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$. Solve $\mathbf{M}\mathbf{z}_0 = \mathbf{r}_0$. $\rho_0 = \mathbf{z}_0^T \mathbf{r}_0$.
Set $\mathbf{p}_1 = \mathbf{z}_0$.
for $k = 1, 2, \ldots$ **do**
  $\mathbf{q}_k = \mathbf{A}\mathbf{p}_k$.
  $\alpha_k = \rho_{k-1} / \mathbf{p}_k^T \mathbf{q}_k$.
  $\mathbf{x}_k = \mathbf{x}_{k-1} + \alpha_k \mathbf{p}_k$.
  $\mathbf{r}_k = \mathbf{r}_{k-1} - \alpha_k \mathbf{q}_k$.
  Solve $\mathbf{M}\mathbf{z}_k = \mathbf{r}_k$.
  $\rho_k = \mathbf{z}_k^T \mathbf{r}_k$.
  if $\rho_k < tol \cdot \rho_0$ **then**
    return
  endif
  $\beta_k = \rho_k / \rho_{k-1}$.
  $\mathbf{p}_{k+1} = \mathbf{z}_k + \beta_k \mathbf{p}_k$.
endfor

The package `Epetra` provides the data structure for the vectors, matrices and the preconditioner. The class references are available on the internet[2].

Note: To be able to use the `Makefile`, make sure that the following modules are loaded:

```
module load intel mkl open_mpi
```

[1] http://trilinos.sandia.gov/packages/docs/r9.0/packages/aztecoo/doc/html/index.html
[2] http://trilinos.sandia.gov/packages/docs/r9.0/packages/epetra/doc/html/index.html