Example : Iterative Solvers

Demonstrates the usage of virtual topology.

In this example, we demonstrate an application of the cartesian topology by way of a simple elliptic (Laplace) equation.

- Fundamentals The Laplace equation, along with prescribed boundary conditions, are introduced. Finite Difference Method is then applied to discretize the PDE to form an algebraic system of equations.
- Jacobi Scheme A very simple iterative method, known as the Jacobi Scheme, is described. A single-process computer code is shown. This program is written in fortran 90 for its concise but clear array representations. (Parallelism and other improvements will be added to this code as we progress)
- Parallel Jacobi Scheme A parallel algorithm for this problem is discussed. Simple MPI routines, without the invocations of cartesian topology, are inserted into the basic, single-process, code above to form the parallel code.
- SOR Scheme The Jacobi scheme, while simple and hence desirable for demonstration purposes, is impractical for "real" applications due to its slow convergence. Enhancements to the basic technique are introduced which leads to the Successive Over Relaxation (SOR) scheme.
- Parallel SOR Scheme With the introduction of a "red-black" algorithm, we can employ the parallel algoithm used for Jacobi to parallelize the SOR.
- Scalability We show the performance of the code for various number of processes to demonstrate its scalability.
- *Click here* to download a zipped file containing the F90 Jacobi and SOR codes, along with make files and a matlab m-file for plotting the solution.

Fundamentals

First, some basics:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \tag{1}$$

where $u = u(x, y)$ is an unknown scalar potential subjected to the following boundary conditions :

$$\begin{aligned} u(x,0) &= sin(\pi x) \qquad & 0 \le x \le 1 \\ u(x,1) &= sin(\pi x)e^{-\pi} \qquad & 0 \le x \le 1 \\ u(0,y) &= u(1,y) = 0 \qquad & 0 \le y \le 1 \end{aligned} \tag{2}$$

Discretize the equation numerically with centered difference results in an algebraic equation

$$u_{i,j}^{n+1} \simeq \frac{u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n}{4} \ ; \quad i = 1, \ldots, \ m \ ; \ j = 1, \ldots, \ m \tag{3}$$

where n and $n+1$ denote the current and the next time step respectively while $u_{i-1,j}^n$, for instance, represents

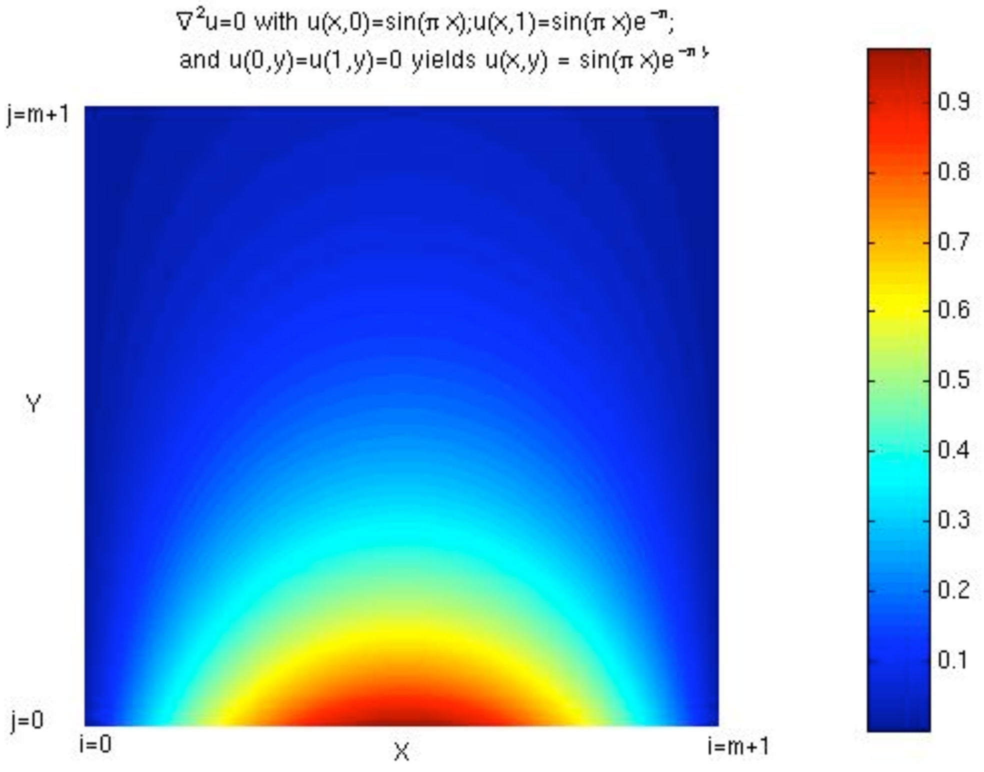
$$\begin{aligned} u_{i-1,j}^n &= u^n(x_{i-1}, y_j) \ ; \qquad i = 1, \ \ldots, m; \quad j = 1, \ \ldots, m \\ &= u^n((i-1)\Delta x, j\Delta y) \end{aligned} \tag{4}$$

and for simplicity, we take $\Delta x = \Delta y = \frac{1}{m+1}$.

Note that the analytical solution for this boundary value problem can easily be verified to be

$$u(x,y) = sin(\pi x)e^{-\pi y} \ ; \qquad 0 \le x \le 1 \ ; \ \ 0 \le y \le 1 \tag{5}$$

and is shown below in a contour plot with x pointing from left to right and y going from bottom to top.



Jacobi Scheme

While numerical techniques abound to solve PDEs such as the Laplace equation, we will focus on the use of an iterative method as it will be shown to be readily parallelizable and lends itself to the opportunity to apply cartesian topology. The simplest of iterative techniques is the Jacobi scheme, which can be stated as follows:

1. Make initial guess for $u_{i,j}$ at all interior points (i,j) for all $i=1:m$ and $j=1:m$.
2. Use Eq. 3 to compute $u^{n+1}_{i,j}$ at all interior points (i,j).
3. Stop if prescribed convergence threshold is reached, otherwise continue on next step.
4. $u^{n}_{i,j} = u^{n+1}_{i,j}$
5. Go to Step 2.

Single process f90 code for the Jacobi Scheme

Parallel Algorithm for the Jacobi Scheme

To enable parallelism, first we need to divvy up works for individual processes; this is known commonly as domain decomposition. Since the governing equation is two-dimensional, typically we have a choice of using a 1D or 2D decomposition. Here, we will focus on a 1D decomposition and defer the discussion of a 2D decomposition for later. Assuming that p processes will be used, we split the computational domain into p horizontal strips, each assigned to one process, along the north-south or y-direction. This choice is made primarily to facilitate a simpler boundary condition (code) implementations.

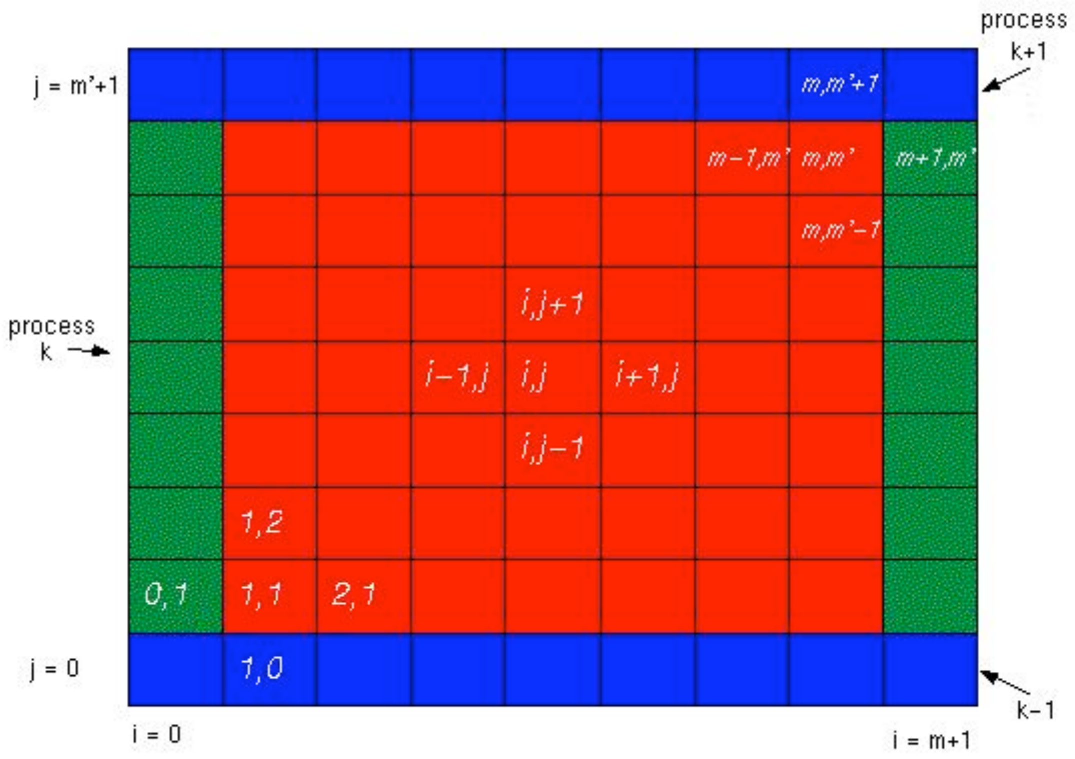
For the obvious reason of better load-balancing, we will divide the amount of work, in this case proportional to the grid size, evenly among the processes *(m x m / p)*. For convenience, we define *m' = m/p* as the number of cells in the y-direction for each process. Next, lets rewrite Eq. 3 for a process *k* as follows:

$$v_{i,j}^{n+1,k} \simeq \frac{v_{i+1,j}^{n,k} + v_{i-1,j}^{n,k} + v_{i,j+1}^{n,k} + v_{i,j-1}^{n,k}}{4}; \qquad (6)$$

$$i = 1, \ldots, m; \; j = 1, \ldots, m'; k = 0, \ldots, p-1$$

where v denotes the local solution corresponding to the process k with $m' = m/p$.

The figure below depicts the grid of a typical process *k* as well as part of adjoining grids of *k-1, k+1*.



- The cells that hold the solution *v* at the new iteration step, *n+1*, as governed by Eq. 6 are represented in red. We will call these *solution cells* or *interior cells*. The terms on the right hand side of Eq. 6, which are known quantities at iteration *n*, span the entire grid system shown.
- The blue cells on the top row represent cells belonging to the first row (*j = 1*) of solution cells of process *k+1* and the blue cells on the bottom row represent the last row (*j = m'*) of solution cells of process *k-1*. It is important to emphasize that the *v* at the blue cells of *k* belong to adjoining processes (*k-1* and *k+1*). They are not accessible to the current process and hence must be "imported", or "received", via MPI message passing routines. Similarly, process *k*'s first and last rows of cells must be "exported", or "sent", to adjoining processes for the same reason that the current process *k* need "imports".
- For *i = 1* and *i = m*, Eq. 6 again requires an extra cell beyond these two locations. These cells contain the prescribed boundary conditions (*u(0,y) = u(1,y) = 0*) and are colored green

to distinguish them from the red and blue cells. Note that no message passing operations is needed for these green cells as they are fixed boundary conditions and are known a priori.

- From the standpoint of process *k*, we consider the blue and green cells as *boundary cells*. Consequently, the range of the strip becomes *(0:m+1,0:m'+1)*. To its green cells we will impose physical boundary conditions while to its blue cells we import *v* from two adjoing processes. With the boundary conditions in place, we can proceed to apply Eq. 6 to all its solution cells. Concurrently, all other processes proceed following the same procedure. It is interesting to note that the grid layout for a typical process *k* is completely analogous to that of the original undivided grid. Whereas the orginal problem has fixed boundary conditions, the problem for a process *k* is subjected to variable boundary conditions.
- These boundary conditions can be stated mathematically as :

$$\begin{aligned}
v^{n,k}_{i,0} &= u(x_i,0) = sin(\pi x_i)\ ; \quad i=0,\ \ldots, m+1; \quad k=0 \\
v^{n,k}_{i,m'+1} &= v^{n,k+1}_{i,1}\ ; \quad i=0,\ \ldots, m+1; \quad k=0 \\
v^{n,k}_{i,0} &= v^{n,k-1}_{i,m'}\ ; \quad i=0,\ \ldots, m+1; \quad 0<k<p-1 \\
v^{n,k}_{i,m'+1} &= v^{n,k+1}_{i,1}\ ; \quad i=0,\ \ldots, m+1; \quad 0<k<p-1 \\
v^{n,k}_{i,0} &= v^{n,k-1}_{i,m'}\ ; \quad i=0,\ \ldots, m+1; \quad k=p-1 \\
v^{n,k}_{i,m'+1} &= u(x_i,1) = sin(\pi x_i)e^{-x}\ ; \quad i=0,\ \ldots, m+1; \quad k=p-1 \\
v^{n,k}_{0,j} &= u(0,y_j) = 0\ ; \quad j=1,\ \ldots, m'; \quad 0\le k\le p-1 \\
v^{n,k}_{m+1,j} &= u(1,y_j) = 0\ ; \quad j=1,\ \ldots, m'; \quad 0\le k\le p-1
\end{aligned} \tag{7}$$

- Note that the interior points of *u* and *v* are related by the following relationship

$$u^n_{i,j+k*m/p} = v^{n,k}_{i,j}\ ; \quad i=1,\ \ldots, m\ ; \quad j=1,\ \ldots, m'\ ; \quad 0\le k\le p-1 \tag{8}$$

Jacobi Parallel Implementation. Note that Cartesian topology is not employed in this implementation but will be used later in the parallel SOR example with the purpose of showing alternative ways to solve this type of problems.

Successive Over Relaxation (SOR)

While the Jacobi iteration scheme is very simple -- and parallelizable -- its slow convergent rate however renders it impractical for any "real world" applications. One way to speed up the convergent rate would be to "over predict" the new solution by linear extrapolation. This leads to the Successive Over Relaxation scheme, or SOR:

1. Make initial guess for $u_{i,j}$ at all interior points *(i,j)*.
2. Define a scalar w_n ($0 < w_n < 2$).
3. Apply Eq. 3 to all interior points *(i,j)* and call it $u'_{i,j}$.
4. $u^{n+1}_{i,j} = w_n u'_{i,j} + (1 - w_n) u^n_{i,j}$
5. Stop if prescribed convergence threshold is reached, otherwise continue on next step.
6. $u^n_{i,j} = u^{n+1}_{i,j}$
7. Go to Step 2.

Note in the above that setting $w_n = 1$ recovers the Jacobi scheme while $w_n < 1$ underrelaxes the solution. Ideally, the choice of w_n would be such that it provides the optimal rate of convergence and is not restricted to a fixed constant. As a matter of fact, an effective choice of w_n, known as the Chebyshev acceleration, is defined as

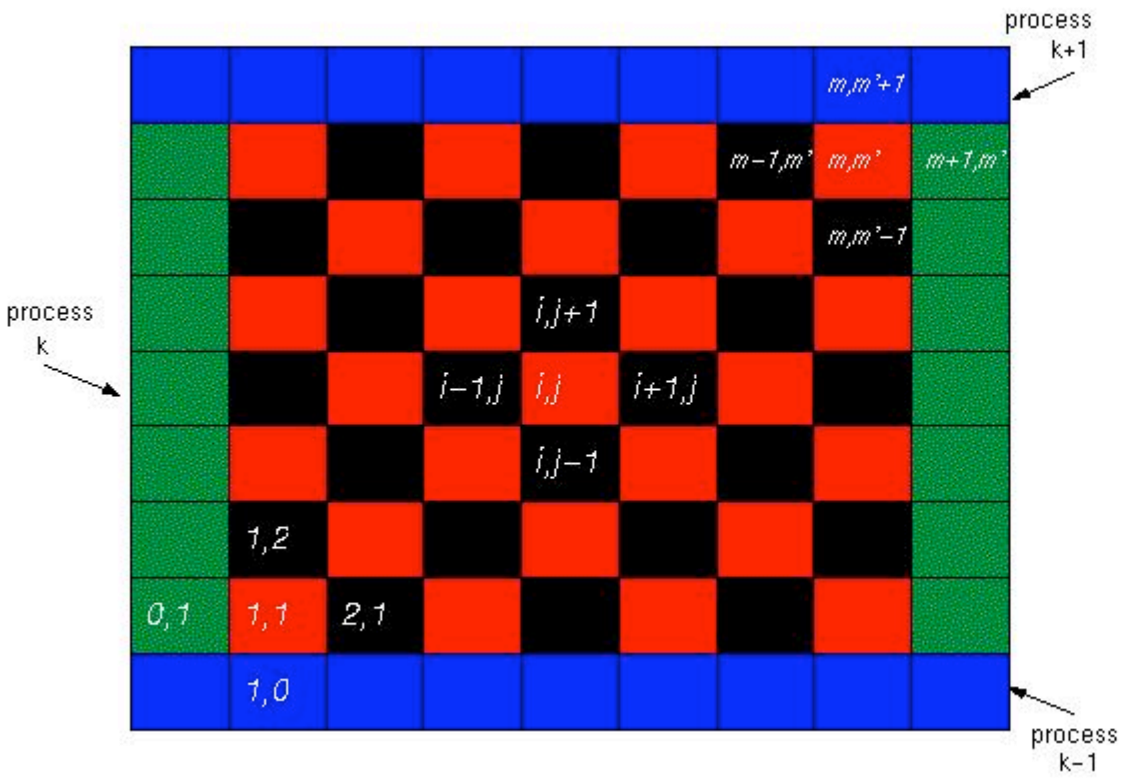
$$\omega_n = \begin{cases} 0 & \text{for } n = 0 \\ \frac{1}{1-\rho^2/2} & \text{for } n = 1 \\ \frac{1}{1-\rho^2\omega_1/4} & \text{for } n = 2 \\ \frac{1}{1-\rho^2\omega_{q-1}/4} & \text{for } n = q > 2 \end{cases}$$

where $\rho = 1 - \left(\frac{\pi}{2(m+1)}\right)^2$ is the spectral radius.

For details, see Chapter 19.5, Relaxation Methods for Boundary Value Problems, the *Numerical Recipe* .

A Parallel SOR Red-black Scheme

To further speed up the rate of convergence, we could use *u* at time level n+1 for any or all terms on the right hand side of Eq. 6 as soon as they become available. This is the essence of the Gauss-Seidel scheme. This requires that the solution be updated cell-by-cell via `do` loop in order to make use of the most recent solution as soon as they become available. Alternatively, we will explore a conceptually similar red-black scheme which will work equally well for an array-based language such as F90 as a scalar-based language such as C or F77. This scheme can best be understood visually by painting the interior cells alternately in red and black to yield a Checkerboard-like pattern:



With this red-black group identification strategy and upon applying the five-point finite-difference stencil to a point *(i,j)* located at a red cell, it is immediately apparent that the solution at the red cell depends only on its four immediate black neighbors to the north, east, west, and south by virtue of Eq. 6. On the contrary, a point *(i,j)* located at a black cell depends only on its north, east, west, and south red neighbors. In other words, the finite-difference stencil in Eq. 6 effects an uncoupling of the solution at interior cells such that solution at the red cells depend only on solution at the black cells and vice versa. In a typical iteration, if we first perform an update on all red *(i,j)* cells, then when we perform the remaining update on black *(i,j)* cells, we could use the red cells that have just been updated. Otherwise, everything that we described about the parallel Jacobi scheme applies equally well here; *i.e.,* the green cells represent the physical boundary conditions while the solutions from first and last rows of the grid of each process are deposited into the blue cells of respective process grids to be used as the remaining boundary conditions.

parallel F90 SOR code. Cartesian topology is employed in this implementation.

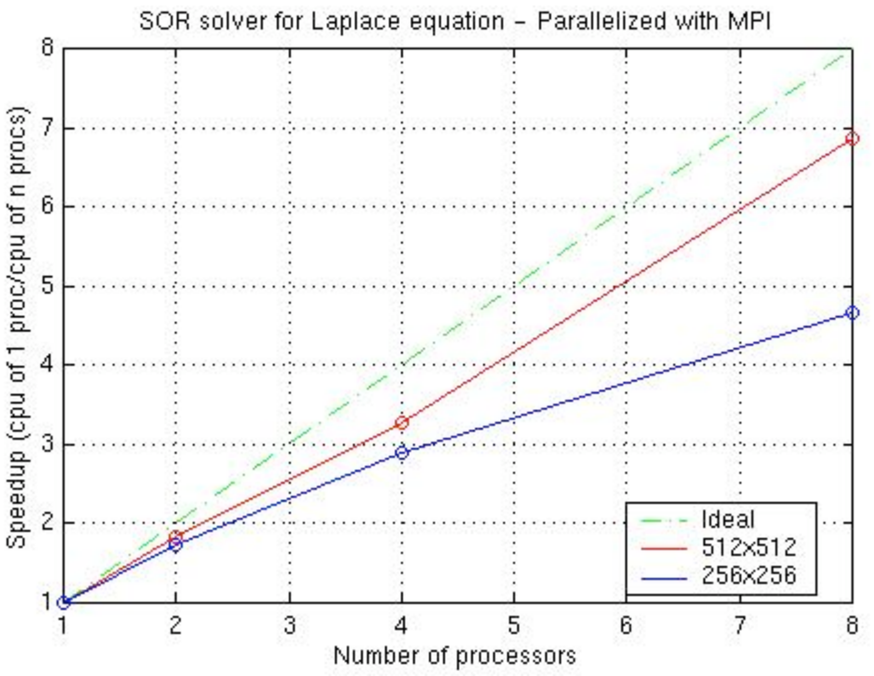
A few comments ...

- Due to array syntax in f90, update per iteration must be done simultaneously. *Do* loops on the other hand permits update cell by cell and hence follows Gauss-Seidel more closely.
- Had we chosen a 2D decomposition, not only that the boundary conditions would be more cumbersome to implement, the interior strips (*i.e.,* k=1, ...,p-2) would have had all blue cells on their borders which would mean additional message passing and hence hinders overall performance.

Scalability Plot of SOR

Here is a plot showing the scalability of the MPI implementation of the Laplace equation using

SOR on an SGI Origin 2000 shared-memory multiprocessor.


SOR solver for Laplace equation - Parallelized with MPI
Speedup (cpu of 1 proc/cpu of n procs)
Number of processors
Ideal
512x512
256x256