Domain Decomposition Example

This project will exemplify a domain decomposition method for finding solutions to a model boundary value problem. The model problem is

$$(BVP) \qquad \frac{d^2u}{dx^2} = f(x), \text{ for } x \in (x_l, x_r) \text{ with } u(x_l) = 0 \text{ and } u(x_r) = 0.$$

f above is a given function of x.

Break the interval $[x_l, x_r]$ into $I+1$ uniform grid points

$$x_i = x_l + i \cdot \Delta x, \quad \Delta x = \frac{x_r - x_l}{I}, \quad 0 \le i \le I.$$

At grid point x_i the exact solution $u(x_i)$ is approximated by u_i. Consider the second centered difference operator

$$D^2 u_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta x^2},$$

and finite difference scheme

$$(FDS) \qquad D^2 u_i = f(x_i), \text{ for } 1 \le i \le I-1 \text{ with } u_0 = 0 \text{ and } u_I = 0.$$

It's easy to see this scheme yields a second order accurate approximation of the BVP's exact solution.

We will solve this coupled FDS system by the simplest iterative method, artificial time relaxation

$$u_i^{n+1} = u_i^n + \Delta t \left(D^2 u_i^n - f(x_i) \right), \quad n = 0, 1, 2, \ldots.$$

u^0 is an arbitrary initial guess and $u = \lim_{n\to\infty} u^n$ is the sought for FDS solution. We'll see below that this iteration will converge provided the artificial time parameter is restricted to satisfy $\Delta t < \frac{1}{2}\Delta x^2$.

Let $e^n = u - u^n$ denote the iteration error. Clearly this evolves according to

$$(ERR) \qquad e_i^{n+1} = e_i^n + \Delta t \, D^2 e_i^n.$$

It's not hard to compute the eigenvectors and eigenvalues to the discrete D^2 operator

$$D^2 (r_k)_i = \lambda_k (r_k)_i$$

enumerated $k = 1, 2, \ldots, I-1$, where

$$(r_k)_i = \sin(k\pi i/I), \qquad \lambda_k = -\frac{4}{\Delta x^2} \sin^2(k\pi/2I).$$

From these, the rate of convergence can be read off. Let e_k^n denote the kth eigencomponent of the iteration error at iteration step n and find that $e_k^n = (1+\Delta t \lambda_k)^n \, e_k^0$. For convenience, let's take $\Delta t = \frac{1}{4}\Delta x^2$ to write the iteration error decay as

$$e_k^n = (\rho_k)^n \, e_k^0 \quad \text{where} \quad \rho_k = 1 - \sin^2(k\pi/2I).$$

Observe that the high frequency components of the error, i.e. $k \sim I$, decay very rapidly, i.e. $\rho_k \sim 0$. However, for the low frequency components, $k \sim 1$, we have

$$k \sim 1 \quad \Rightarrow \quad \rho_k \sim 1 - \left(\frac{k\pi}{2I}\right)^2 \approx 1.$$

That is, smoothly varying components of the iteration error decay slowly, and the rate becomes worse with increasing number of grid points.

Of course, nobody would use iteration to solve this very simple one dimensional model problem. However, this example does shed light on a critical issue when employing iterative methods for solving elliptic BVP's. Even in situations where they make sense (for example in two or three dimensions), they may (and often do) converge slowly.

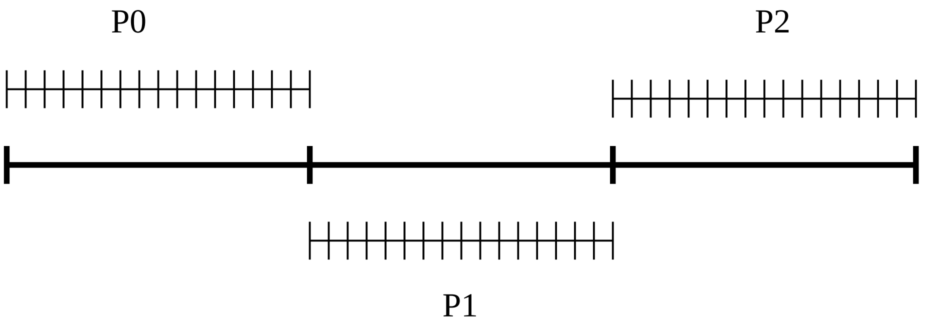
Here's a stub of C-code utilizing artificial time to solve our FDS.

```
float F(float x) {...}
...
int ...;
float xl, xr, dx, dx2, odx2;
float *u0 = NULL, *u1 = NULL, *utmp;
...
xl = XL; xr = XR;
...
u0 = (float*)malloc( (I+1)*sizeof(float) );
u1 = (float*)malloc( (I+1)*sizeof(float) );
u0[0] = u1[0] = UL; u0[I] = u1[I] = UR;
...
dx = (xr-xl)/I; dx2 = dx*dx; odx2 = 1.0/dx2
...
for( n = 0; n < NITS; n++ ) {
  float dt = dx2/4, x;
  x = xl+dx;
  for( i = 1; i < I; i++ ) {
    u1[i] = u0[i] + dt*( (u0[i+1]-2*u0[i]+u0[i-1])*odx2 - F(x) );
    x += dx;
  }
  utmp = u0;
  u0 = u1;
  u1 = utmp;
}
```

Domain Decomposition

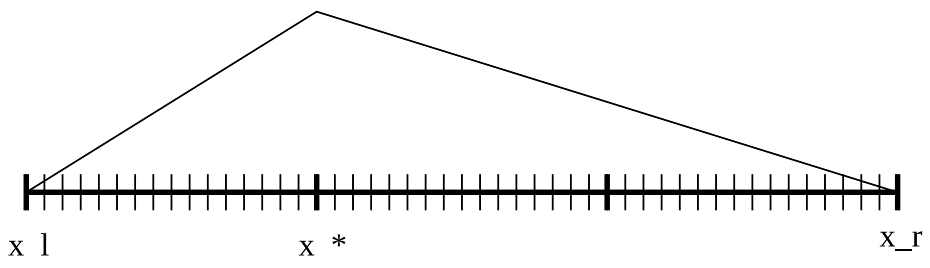
What we'll do is split the physical domain into np (number of processes) pieces with each piece working in its own process with its own local grid; see the figure below. Suppose the physical domain for the full BVP runs from $x = XL$ to $x = XR$, and each local grid has $I+1$ grid points with a one point overlap. Then, each local grid has physical width

$dX = (XR - XL)/np$, $\Delta x = dX/I$, and on process $id \in [0, np-1]$ its local x ranges from $xl = XL + dX * id$ to $xr = xl + dX$.



This figure exemplifies the full grid split into three local grids with each in its own process: P_0, P_1 and P_2. Here there is a one point overlap at each process interface.

Here are the basics of our domain decomposition algorithm. We iterate a certain number of times on each local grid in a manner completely decoupled from all other local grids; i.e. the local index i ranges from $1 \le i < I$ as in the C-stub above. This will drive the residual, $R_i^n \equiv D^2 u_i^n - f(x_i)$, to a small value at all points except at the overlapping end points. Even though the residual may be (close to) zero at all but a few interface points, the actual iteration error is more than likely quite large.



Let v be the piecewise linear and continuous function depicted in the graph above; $v(x_l) = 0$, $v(x_r) = 0$ and $v(x_*) = 1$ where x_* is the interface point between P_0 and P_1. One easily sees that $D^2 v(x_i) = 0$ at every x_i except when $x_i = x_*$.

To correct this error due to decoupling, consider the function $v(x)$ depicted in the figure directly above. That is

$$v(x) = \begin{cases} (x - x_l)/(x_* - x_l) & \text{if } x_l \le x \le x_* \\ (x_r - x)/(x_r - x_*) & \text{if } x_* < x \le x_r \end{cases} \quad \Rightarrow \quad v(x_*) = 1.$$

where x_* denotes the overlap point shared by process P_0 and P_1. An easy calculation reveals at $i = i_*$

$$D^2 v(x_{i_*}) = \frac{1}{\Delta x}\left(\frac{(0-1)}{(x_r - x_*)} - \frac{(1-0)}{(x_* - x_l)}\right) = -\frac{1}{\Delta x}\left(\frac{(x_r - x_l)}{(x_r - x_*)(x_* - x_l)}\right),$$

and so for every i

$$D^2 v(x_i) = \begin{cases} -\frac{(x_r - x_l)}{\Delta x (x_r - x_*)(x_* - x_l)} & \text{if } x_i = x_* \\ 0 & \text{if } x_i \neq x_*. \end{cases}$$

From this we deduce the exact discrete solution to

$$D^2 \epsilon_i = \delta_{i,i_*} \quad \Rightarrow \quad \epsilon_i = -\frac{\Delta x (x_r - x_*)(x_* - x_l)}{(x_r - x_l)} v(x_i),$$

where $\delta_{i,j}$ as usual denotes the Kronecker delta.

To finish the discussion on how to correct the decoupling error, assume the residual has been driven to zero at interior points. At a typical interface, x_*, write the exact FDS solution as $u_i = u_i^n + e_i^n$, where u_i^n is the current iteration value and e_i^n is the iteration error. So

$$0 = D^2(u_i^n + e_i^n) - f(x_i) \quad \Rightarrow \quad D^2(e_i^n) = -R_i^n = -R_{i_*}^n \delta_{i,i_*} \text{ (our assumption).}$$

Solve this exactly as outlined above for e_i^n and correct the current state by $u_i^n \to u_i^n + e_i^n$. This step is repeated for every local grid interface.

MPI Implementation

Now you don't want to move a bunch of data from one process to another. Here's what I suggest you do. Let the process whose right endpoint is an interface handle that interface. You'll first want to compute all interface residuals something like this.

```
if( myid > 0 ) {
  wait
  send local u[1] to myid-1.
}
if( myid < np-1 ) {
  recv urr ( = u[1] ) from myid+1.
  Use local u[I-1], u[I] and recv'd urr values to
  compute interface residual, say R, at local xr.
}
```

Next, broadcast correction endpoint values to everybody who needs them, and receive correction endpoint values from everybody who makes them. Let `float corr[2]` be a send/recv buffer and `float e[2]` a buffer to hold accumulated endpoint errors.

```
if( myid < np-1 ) {
  This process is handling a correction.
  Therefore, it already computed the needed interface residual R.
  for( id = 0; id < myid; id++ ) {
    See figure above. These are for v(x) when x < x_*.
    corr[0] = left endpoint value of correction for process id.
    corr[1] = right endpoint value of correction for process id.
    wait
```

```
      send two word corr[] to id.
    }
    for( id = myid+1; id < np; id++ ) {
      See figure above. These are for v(x) when x > x_*.
      corr[0] = left endpoint value of correction for process id.
      corr[1] = right endpoint value of correction for process id.
      wait
      send two word corr[] to id.
    }

    Never send stuff to yourself.
    These corrections are also for v(x) when x < x_*.
    e[0] = correction for my local xl.
    e[1] = correction for my local xr.
  } else {
    Process np-1 doesn't do any interface correction stuff.
    e[0] = 0.0;
    e[1] = 0.0;
  }
  Remember, process np-1 doesn't send anything.
  for( id = 0; id < np-1; id++ ) {
    Never recv stuff from yourself.
    myid was done already when e[] was initialized above.
    if( id == myid ) continue;
    recv two word corr[] from id.
    e[0] += corr[0];
    e[1] += corr[1];
  }
```

The last step is to linearly interpolate `e[0]` and `e[1]` and use the interpolation to update all $(0 \le i \le I)$ local values of u_i^n.