Domain Decomposition Methods

1. Gauss-Seidel Relaxation
   - an iterative method for solving linear systems
   - a parallel Gauss-Seidel with OpenMP
2. Solving the Heat Equation
   - applying a time stepping method to a PDE
   - domain decomposition
3. Solving the Heat Equation with PETSc
   - The Portable, Extensible Toolkit for Scientific Computation

MCS 572 Lecture 30
Introduction to Supercomputing
Jan Verschelde, 4 November 2024

Domain Decomposition Methods

1. Gauss-Seidel Relaxation
   - an iterative method for solving linear systems
   - a parallel Gauss-Seidel with OpenMP

2. Solving the Heat Equation
   - applying a time stepping method to a PDE
   - domain decomposition

3. Solving the Heat Equation with PETSc
   - The Portable, Extensible Toolkit for Scientific Computation

fixed point formula for Gauss-Seidel relaxation

The fixed point formula for $A\mathbf{x} = \mathbf{b}$ where $A = L + D + U$,

- L is strict lower triangular, $L = [a_{i,j}]$, $i > j$, 0 otherwise
- D is diagonal, $D = [a_{i,j}]$, $i = j$, 0 otherwise
- U is strict upper triangular, $U = [a_{i,j}]$, $i < j$, 0 otherwise

$$\begin{aligned} A\mathbf{x} = \mathbf{b} \quad & \Leftrightarrow \quad (L + D + U)\mathbf{x} = \mathbf{b} \\ & \Leftrightarrow \quad (L + D)\mathbf{x} + U\mathbf{x} = \mathbf{b} \\ & \Leftrightarrow \quad (L + D)\mathbf{x} = \mathbf{b} - U\mathbf{x} \end{aligned}$$

Observe that $L + D$ is lower triangular.
We apply forward substitution in each step.

the formulas for Gauss-Seidel relaxation

We want to solve $A\mathbf{x} = \mathbf{b}$ for $A \in \mathbb{R}^{n \times n}$, $\mathbf{b} \in \mathbb{R}^n$, for ***very large*** n.

Writing the method of Jacobi componentwise:

$$x_i^{(k+1)} := x_i^{(k)} + \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{n} a_{i,j} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n$$

we observe that we can already use $x_j^{(k+1)}$ for $j < i$.

This leads to the following formulas

$$x_i^{(k+1)} := x_i^{(k)} + \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^{i-1} a_{i,j} x_j^{(k+1)} - \sum_{j=i}^{n} a_{i,j} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n.$$

the Gauss-Seidel method

Writing the formulas as an algorithm:

Input: A, $\mathbf{b}$, $\mathbf{x}^{(0)}$, ϵ, N.
Output: $\mathbf{x}^{(k)}$, k is the number of iterations done.

```
for $k$ from 1 to $N$ do
    for $i$ from 1 to $n$ do
        $\Delta x_i := b_i$
        for $j$ from 1 to $i-1$ do
            $\Delta x_i := \Delta x_i - a_{i,j} x_j^{(k+1)}$
        for $j$ from $i$ to $n$ do
            $\Delta x_i := \Delta x_i - a_{i,j} x_j^{(k)}$
        $\Delta x_i := \Delta x_i / a_{i,i}$
        $x_i^{(k+1)} := x_i^{(k)} + \Delta x_i$
    exit when $(||\Delta \mathbf{x}|| \leq \epsilon)$
```

loop fusing

The method of Gauss-Seidel is an *in-place method*:
old values are overwritten by new ones as soon as computed.

The two loops in

```
for $j$ from 1 to $i - 1$ do
    $\Delta x_i := \Delta x_i - a_{i,j} x_j^{(k+1)}$
for $j$ from $i$ to $n$ do
    $\Delta x_i := \Delta x_i - a_{i,j} x_j^{(k)}$
```

are fused into one loop:

```
for $j$ from 1 to $n$ do
    $\Delta x_i := \Delta x_i - a_{i,j} x_j$
```

C code for the Gauss-Seidel method

```
void run_gauss_seidel_method
 ( int n, double **A, double *b,
   double epsilon, int maxit,
   int *numit, double *x )
/*
 * Runs the  method of Gauss-Seidel for A*x = b.
 *
 * ON ENTRY :
 *   n         the dimension of the system;
 *   A         an n-by-n matrix A[i][i] /= 0;
 *   b         an n-dimensional vector;
 *   epsilon   accuracy requirement;
 *   maxit     maximal number of iterations;
 *   x         start vector for the iteration.
 *
 * ON RETURN :
 *   numit     number of iterations used;
 *   x         approximate solution to A*x = b. */
```

code for run_gauss_seidel_method

```
{
   double *dx = (double*) calloc(n,sizeof(double));
   int i,j,k;
   for(k=0; k<maxit; k++)
   {
      double sum = 0.0;
      for(i=0; i<n; i++)
      {
         dx[i] = b[i];
         for(j=0; j<n; j++)
            dx[i] -= A[i][j]*x[j];
         dx[i] /= A[i][i]; x[i] += dx[i];
         sum += ( (dx[i] >= 0.0) ? dx[i] : -dx[i]);
      }
      printf("%4d : %.3e\n",k,sum);
      if(sum <= epsilon) break;
   }
   *numit = k+1; free(dx);
}
```

the test system

For the dimension n, we consider the diagonally dominant system:

$$\begin{bmatrix} n+1 & 1 & \cdots & 1 \\ 1 & n+1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & n+1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} 2n \\ 2n \\ \vdots \\ 2n \end{bmatrix}.$$

The exact solution is $\mathbf{x}$: for $i = 1, 2, \ldots, n$, $x_i = 1$.

We start the iterative method at $\mathbf{x}^{(0)} = \mathbf{0}$.

Values for the parameters:

- $\epsilon = 10^{-4}$ as the tolerance on the accuracy; and
- $N = 2n^2$ for the maximum number of iterations.

running on the test system

```
$ time ./gauss_seidel 1000
   0 : 1.264e+03
   1 : 3.831e+02
   2 : 6.379e+01
   3 : 1.394e+01
   4 : 3.109e+00
   5 : 5.800e-01
   6 : 1.524e-01
   7 : 2.521e-02
   8 : 7.344e-03
   9 : 1.146e-03
  10 : 3.465e-04
  11 : 5.419e-05
computed 12 iterations     <----- 8407 with Jacobi
error : 1.477e-05

real    0m0.069s           <----- 0m42.411s
user    0m0.063s           <----- 0m42.377s
sys     0m0.005s           <----- 0m0.028s
```

Domain Decomposition Methods

1. Gauss-Seidel Relaxation
   - an iterative method for solving linear systems
   - a parallel Gauss-Seidel with OpenMP

2. Solving the Heat Equation
   - applying a time stepping method to a PDE
   - domain decomposition

3. Solving the Heat Equation with PETSc
   - The Portable, Extensible Toolkit for Scientific Computation

granularity considerations

The method of Jacobi is suitable for strip partitioning of the (dense) matrix and in a parallel distributed memory implementation, every processor can keep its own portion of the solution vector **x**.

The Gauss-Seidel method makes the new x_i directly available which leads to communication overhead on distributed memory computers.

In a parallel shared memory implementation, consider:

1. Threads compute inner products of matrix rows with **x**.
2. Each Δx_i is updated in a critical section.

many threads compute one inner product

For example, three threads, assuming n is divisible by 3, compute:

$$\left[a_{i,1} \cdots a_{i,n/3} \,\middle|\, a_{i,n/3+1} \cdots a_{i,2n/3} \,\middle|\, a_{i,2n/3+1} \cdots a_{i,n} \right] \left[\begin{array}{c} x_1 \\ \vdots \\ x_{n/3} \\ \hline x_{n/3+1} \\ \vdots \\ x_{2n/3} \\ \hline x_{2n/3+1} \\ \vdots \\ x_n \end{array} \right]$$

Each thread has its own variable to accumulate its portion of the inner product.

using p threads

```
void run_gauss_seidel_method
 ( int p, int n, double **A, double *b,
   double epsilon, int maxit,
   int *numit, double *x )
{
   double *dx;
   dx = (double*) calloc(n,sizeof(double));
   int i,j,k,id,jstart,jstop;

   int dnp = n/p;
   double dxi;

   for(k=0; k<maxit; k++)
   {
      double sum = 0.0;
      for(i=0; i<n; i++)
      {
```

the parallel region

Threads collaborate at making one inner product.

```
dx[i] = b[i];
#pragma omp parallel \
   shared(A,x) \
   private(id,j,jstart,jstop,dxi)
{
   id = omp_get_thread_num();
   jstart = id*dnp;
   jstop = jstart + dnp;
   dxi = 0.0;
   for(j=jstart; j<jstop; j++)
      dxi += A[i][j]*x[j];
   #pragma omp critical
      dx[i] -= dxi;
}
```

after the parallel region

The update instructions

```
        dx[i] /= A[i][i];
        x[i] += dx[i];
        sum += ( (dx[i] >= 0.0) ? dx[i] : -dx[i]);
```

are executed after each parallel region.

This ensures the synchronization and the execution of the stop test:

```
    if(sum <= epsilon) break;
```

running times on 12-core Intel X5690, 3.47 GHz

```
$ time ./gauss_seidel_omp n p
```

p	n	real	user	sys	speedup
1	10,000	7.165s	6.921s	0.242s	
	20,000	28.978s	27.914s	1.060s	
	30,000	1m 6.491s	1m 4.139s	2.341s	
2	10,000	4.243s	7.621s	0.310s	1.689
	20,000	16.325s	29.556s	1.066s	1.775
	30,000	36.847s	1m 6.831s	2.324s	1.805
5	10,000	2.415s	9.440s	0.420s	2.967
	20,000	8.403s	32.730s	1.218s	3.449
	30,000	18.240s	1m11.031s	2.327s	3.645
10	10,000	2.173s	16.241s	0.501s	3.297
	20,000	6.524s	45.629s	1.521s	4.442
	30,000	13.273s	1m29.687s	2.849s	5.010

Domain Decomposition Methods

1. Gauss-Seidel Relaxation
   - an iterative method for solving linear systems
   - a parallel Gauss-Seidel with OpenMP

2. Solving the Heat Equation
   - applying a time stepping method to a PDE
   - domain decomposition

3. Solving the Heat Equation with PETSc
   - The Portable, Extensible Toolkit for Scientific Computation

the heat equation

The heat or diffusion equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{\partial u}{\partial t}$$

models the temperature distribution $u(x, y, t)$
evolving in time t for (x, y) in some domain.

Related Partial Differential Equations (PDEs) are

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0 \quad \text{and} \quad \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y),$$

respectively called the Laplace and Poisson equations.

initial and boundary conditions

For $t > 0$, we consider the domain of

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{\partial u}{\partial t}$$

to be $[0,1] \times [0,1]$, so $0 \leq x \leq 1$ and $0 \leq y \leq 1$.

For numerical computations, we must specify this problem with

- one initial condition: $u(x,y,0) = f(x,y)$; and
- four boundary conditions:
  1. $u(0,y,t) = g_1(y)$,
  2. $u(1,y,t) = g_2(y)$,
  3. $u(x,0,t) = g_3(x)$, and
  4. $u(x,1,t) = g_4(x)$.

It suffices to know f, g_1, g_2, g_3, g_4 at the grid points.

discretization of the derivatives

At a point (x_0, y_0, t_0), we have

$$\left.\frac{\partial u}{\partial x}\right|_{(x_0,y_0,t_0)} = \lim_{h\to 0} \underbrace{\frac{u(x_0+h, y_0, t_0) - u(x_0, y_0, h)}{h}}_{u_x(x_0,y_0,t_0)}$$

so for positive $h \approx 0$, $u_x(x_0, y_0, t_0) \approx \left.\frac{\partial u}{\partial x}\right|_{(x_0,y_0,t_0)}$.

For the second derivative we use the finite difference $u_{xx}(x_0, y_0, t_0)$

$$\begin{aligned} &= \frac{1}{h}\left(\frac{u(x_0+h, y_0, t_0) - u(x_0, y_0, t_0)}{h} - \frac{u(x_0, y_0, t_0) - u(x_0-h, y_0, t_0)}{h} \right) \\ &= \frac{u(x_0+h, y_0, t_0) - 2u(x_0, y_0, t_0) + u(x_0-h, y_0, t_0)}{h^2}. \end{aligned}$$

time stepping

$$
\begin{aligned}
u_t(x_0, y_0, t_0) &= \frac{u(x_0, y_0, t_0 + h) - u(x_0, y_0, t_0)}{h} \\
u_{xx}(x_0, y_0, t_0) &= \frac{u(x_0 + h, y_0, t_0) - 2u(x_0, y_0, t_0) + u(x_0 - h, y_0, t_0)}{h^2} \\
u_{yy}(x_0, y_0, t_0) &= \frac{u(x_0, y_0 + h, t_0) - 2u(x_0, y_0, t_0) + u(x_0, y_0 - h, t_0)}{h^2}
\end{aligned}
$$

Then the equation $\dfrac{\partial u}{\partial t} = \dfrac{\partial^2 u}{\partial x^2} + \dfrac{\partial^2 u}{\partial y^2}$ becomes

$$
\begin{aligned}
u(x_0, y_0, t_0 + h) &= u(x_0, y_0, t_0) \\
&+ (1/h)\ [\ u(x_0 + h, y_0, t_0) + u(x_0 - h, y_0, t_0) \\
&+ \quad u(x_0, y_0 + h, t_0) + u(x_0, y_0 - h, t_0) - 4u(x_0, y_0, t_0)\ ]
\end{aligned}
$$

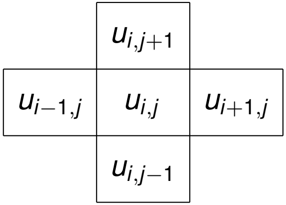
Locally, the error of this approximation is $O(h^2)$.

synchronous iterations on a grid

For $(x, y) \in [0,1] \times [0,1]$, the division of $[0,1]$ in n equal subintervals, with $h = 1/n$, leads to a grid $(x_i = ih, y_j = jh)$, for $i = 0, 1, \ldots, n$ and $j = 0, 1, \ldots, n$.

For t, we use the same step size h: $t_k = kh$.
Denote $u_{i,j}^{(k)} = u(x_i, y_j, t_k)$, then

$$u_{i,j}^{(k+1)} = u_{i,j}^{(k)} + \frac{1}{h} \left[u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)} - 4u_{i,j}^{(k)} \right].$$



In every step, we update $u_{i,j}$ based on $u_{i-1,j}$, $u_{i+1,j}$, $u_{i,j-1}$, and $u_{i,j+1}$.

iterative solving of linear systems

The formulas lead directly to the following algorithm:

```
for $k = 1, 2, \ldots, N$ do
    for $i = 0, 1, \ldots, n$ do
        for $j = 0, 1, \ldots, n$ do
            $u_{i,j}^{(k+1)} = u_{i,j}^{(k)} + \frac{1}{h} \left[ u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)} - 4u_{i,j}^{(k)} \right]$.
```

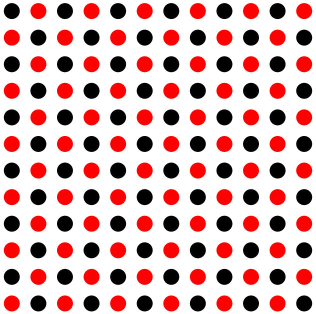
The above loops are similar to Jacobi's method.

Using the most recent values, as in the Gauss-Seidel method, leads to faster convergence.

For this problem, there is a specific ordering that is better suited.

red-black ordering

We divide the grid in red and black points:



The computation is organized in two phases:

1. update all black points simultaneously; and then
2. update all red points simultaneously.

Domain Decomposition Methods

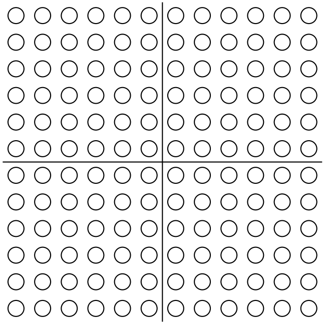
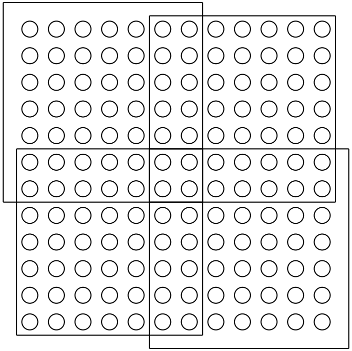
1. Gauss-Seidel Relaxation
   - an iterative method for solving linear systems
   - a parallel Gauss-Seidel with OpenMP

2. Solving the Heat Equation
   - applying a time stepping method to a PDE
   - domain decomposition

3. Solving the Heat Equation with PETSc
   - The Portable, Extensible Toolkit for Scientific Computation

domain decomposition

We can decompose a domain in strips,
but then there are n/p boundaries that must be shared.
To reduce the overlapping, we partition in squares:

Then the boundary elements are proportional to $n/\sqrt{p}$.

comparing communication costs

In a square partition, every square has 4 edges, whereas a strip has only 2 edges. For the communication cost, we multiply by 2 because for every send there is a receive.

Comparing the communication cost for a strip partitioning

$$t_{\rm comm}^{\rm strip} = 4 \left(t_{\rm startup} + n t_{\rm data} \right)$$

to the communication cost for a square partitioning (for $p \geq 9$):

$$t_{\rm comm}^{\rm square} = 8 \left(t_{\rm startup} + \frac{n}{\sqrt{p}} t_{\rm data} \right).$$

A strip partition is best if the startup time is large and if we have only very few processors.

If the startup time is low, and for $p \geq 4$, a square partition starts to look better.

some numerical considerations

The discretization of the heat equation is the simplest one.

- The explicit forward difference method is conditionally stable: in order for the method to converge, the step size in time depends on the step size in space.
- Methods that are unconditionally stable are implicit and require the solving of a linear system in each time step.

Domain Decomposition Methods

1. Gauss-Seidel Relaxation
   - an iterative method for solving linear systems
   - a parallel Gauss-Seidel with OpenMP
2. Solving the Heat Equation
   - applying a time stepping method to a PDE
   - domain decomposition
3. Solving the Heat Equation with PETSc
   - The Portable, Extensible Toolkit for Scientific Computation

PETSc

PETSc = The Portable, Extensible Toolkit for Scientific Computation.

- PETSc provides data structures and routines for large-scale application codes on parallel (and serial) computers, using MPI.
- Support for Fortran, C, C++, Python, and MATLAB (serial).
- Free and open source, available at `http://petsc.org`.
- Part of the ACTS (Advanced CompuTational Software) Collection.

suggested reading

- Ronald F. Boisvert, L. A. Drummond, Osni A. Marques: **Introduction to the special issue on the Advanced CompuTational Software (ACTS) collection.** *ACM TOMS* 31(3):281–281, 2005. Special issue on the Advanced CompuTational Software (ACTS) Collection.

- Visit `https://petsc.org` and browse the documentation.

- Read **The PETSc Community Is the Infrastructure** by Mark Adams, Satish Balay, Oana Marin, Lois Curfman McInnes, Richard Tran Mills, Todd Munson, Hong Zhang, Junchao Zhang, Jed Brown, Victor Eijkhout, Jacob Faibussowitsch, Matthew Knepley, Fande Kong, Scott Kruger, Patrick Sanan, Barry F. Smith, Hong Zhang. `https://arxiv.org/abs/2201.00967`

Summary + Exercises

We covered §6.3.2 in the book of Wilkinson and Allen, see also §11.4.

Exercises:

1. Take the running times of the OpenMP version of the method of Gauss-Seidel and compute the efficiency for each of the 9 cases. What can you conclude about the scalability?
2. Use MPI to write a parallel version of the method of Gauss-Seidel. Compare the speedups with the OpenMP version.
3. Run an example of the PETSc tutorials collection with an increasing number of processes to investigate the speedup.
4. Cellular automata (e.g.: Conway's game of life) are synchronized computations. Discuss a parallel implementation of Conway's game of life and illustrate your discussion with a computation.