

Synchronous and Asynchronous Jacobi Iterative Solvers in Serial, Threaded, MPI, and OpenMP Implementations

CSI702 SEMESTER PROJECT
BOB SORENSEN

Overview: Jacobi Iteration

- For a linear system, expressed as $\mathbf{Ax}=\mathbf{b}$, where \mathbf{A} is an $n \times n$ square matrix, and \mathbf{x} and \mathbf{b} are n -vectors, if the values of \mathbf{A} and \mathbf{b} are known, the appropriate values for \mathbf{x} —the solution to the set of linear equations—can be solved by the Jacobi iteration.
- Specially, given that

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}.$$

- Then \mathbf{A} can be decomposed into two separate $n \times n$ matrices \mathbf{D} and \mathbf{R} such that \mathbf{D} is the diagonal component of \mathbf{A} , and \mathbf{R} is the remainder of \mathbf{A} once \mathbf{D} is removed:

$$A = D + R \quad \text{where} \quad D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}, \quad R = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}$$

Overview: Jacobi Iteration

- And using that $\mathbf{A} = \mathbf{D} + \mathbf{R}$, the original system of linear equations, $\mathbf{Ax}=\mathbf{b}$, can be rewritten as:

$$(D + R)\mathbf{x} = \mathbf{b}$$

$$D\mathbf{x} + R\mathbf{x} = \mathbf{b}$$

$$D\mathbf{x} = \mathbf{b} - R\mathbf{x}$$

$$\mathbf{x} = D^{-1}(\mathbf{b}-R\mathbf{x}) \quad (1)$$

- The Jacobi method uses an iterative technique to solve the left hand side of this expression for \mathbf{x} , using previous values for \mathbf{x} on the right hand side—starting with an initial guess of $\mathbf{x}=[0,0,\dots,0]$. Analytically, this may be written as: $\mathbf{x}^{(k+1)} = D^{-1}(\mathbf{b} - R\mathbf{x}^{(k)})$.

- **The cool part:**

Note that equation (1) calls for computing the inverse of \mathbf{D} , not \mathbf{A} . But, because of the way we have defined \mathbf{D} , this is simple because \mathbf{D} is a diagonal matrix, i.e.

Given $\mathbf{D} = (d_{ij})$, for all $i=j$, 0 elsewhere

Then $\mathbf{D}^{-1} = (1/d_{ij})$, for all $i=j$, 0 elsewhere,

- so what we really end up coding is

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

Jacobi Iterations and Reality-The Fine Details

- However, there is a price to be paid here: the Jacobi method, like many other iterative processes, is not guaranteed to converge to a solution.
 - In order to guarantee convergence, the \mathbf{A} matrix under consideration must be strictly *diagonally dominant*: that is, for each row in \mathbf{A} , the absolute value of each diagonal term must be greater than the sum of absolute values of all other terms in that row, or $|a_{ii}| > \sum_{j \neq i} |a_{ij}|$.
 - Happily, the literature is filled with papers that discuss the process whereby any given non-singular matrix \mathbf{A} can be converted into an equivalent diagonally dominant matrix.
- Finally, as with any iterative process, the solution is arbitrarily reached when the process has converged to some point whereby any additional iteration does not change the ‘answer’ more than some predetermined epsilon. Taking the Euclidean norm of the array \mathbf{x} ;

$$\|\mathbf{x}\| := \sqrt{x_1^2 + \cdots + x_n^2}$$

- Use:

TOLERANCE < abs(|| \mathbf{x}_i || - || \mathbf{x}_{i-1} ||) as stopping criteria

Jacobi: Synchronous and Asynchronous Variations on a Theme

- Based on the discussion above, it is clear that for the traditional Jacobi iteration, the process is done in a highly synchronous manner.
 - That is, the calculations for each iteration are completed for every element in \mathbf{x} before the next iteration can begin.
 - As a result, at each time step, every calculation is using a version of \mathbf{x}_i that may be old and perhaps, already updated elsewhere.
- *Asynchronous* versions always use the most recent calculation for any given \mathbf{x}_i regardless of what time step it was produced.
 - Much less orderly
 - Much less stable
 - Requires MUCH more data transfer (you have been warned)
 - But, perhaps, offers promise of better convergence
- Lots of issues here about 'false' convergence

So it begins.....

- Ultimately, seven version of the Jacobi iteration algorithm were:
 - implemented in C,
 - compiled on the Intel C compiler,
 - run against a series of 512 linear equations (d.d.), and
 - tested on GMU's *cds04.gmu.edu* system.
- They were:
 - Synchronous serial (Both baseline and optimized by the Intel C compiler)
 - Synchronous using pthreads
 - Synchronous using MPI
 - Synchronous using openmp
 - Asynchronous using pthreads
 - Asynchronous using MPI
 - Asynchronous using openmp

The Computation Core:

- This snippet illustrates the heart of the Jacobi iteration: it is simply the C version of the element-based formula for the Jacobi iteration discussed earlier:

```
for(i=0;i<X_SIZE;i++) {
    alpha = 0;
    for(j=0;j<Y_SIZE;j++) {
        if (j != i) {
            alpha = alpha + (a[(i*X_SIZE)+ j] * x_old[j]);
        }
    }
    x_new[i]= (b[i]-alpha) / (a[(i*X_SIZE) + i]);
}
```

- And the stopping conditions

```
while ((x_norm_delta > TOLERANCE) && (counter < MAX_ITERATIONS)) {
```

where the MAX_ITERATION condition keeps the code from running away if there is no convergence .

Results: Good, Bad, and Ugly

<u>Method</u>	<u>Speed-Up</u>	<u>Config</u>	<u>Comment</u>
Sync/Serial/Baseline	1	-O0	No Optimization
Sync/Serial	1.22	-O3	Best Intel C Optimization
Sync/threads	1.54	4-8 threads	18 loops to converge
Sync/MPI	0.27	2 cpus	Degrades rapidly as cpu count increases
Sync/openmp	0.54	2 threads	Constant for 2-32 thread, 18 loops to converge
Async/threads	1.45	2 threads	8 loops, errors for thread counts > 4
Async/MPI	0.0003	4 cpus	2728 times longer than baseline (7m. 22s.)
<u>Async/openmp</u>	<u>0.40</u>	<u>2 threads</u>	<u>8 loops to converge</u>

Summary and Observations

- In the best case, a synchronous pthreads version realized a 1.54 speed up over the baseline serial case. And 26 percent faster than the optimized serial case.
 - *Considering that Intel spends \$3-4 billion on R&D every year—and I don't—that's not too shabby.*
- The slowest implementation, the asynchronous MPI version, ran gloriously slow—over 2700 times slower than the baseline test case—indicating that, perhaps, MPI and shared memory scheme do not mix well.
- All asynchronous versions converged with about 1/2 the iterations needed by their synchronous counterparts (*cool*).
 - But any gains in computational performance were ultimately masked by additional communications overhead (*boo*).

Summary and Observations (cont.)

- Results for the synchronous MPI test reveals the high price one pays for too much data communication at the expense of potential parallel computational capability.
 - The slowest synch MPI test was for 16 cpus, which ran almost 11 times slower than the fastest sync MPI which used only two cpus.
- For the two thread asynchronous pthreads, the code iterated to the right answer. But for increased thread counts, the process iterated to a stable, albeit wrong solution.
 - *The inability to dictate when a particular thread 'relinquishes' control is somewhat annoying.*
- It took approximately 90 seconds to convert the serial code to sync openmp, and about three minutes for the async openmp version.
 - Too bad it didn't run faster