

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

CSI702 Semester Project

Bob Sorensen

Summary:

A system of 512 linear equations was solved using the Jacobi iteration method in various parallel implementations. In the best case, a synchronous pthreads version realized a 1.54 speed up over the baseline serial case. The slowest implementation, the asynchronous MPI version, ran gloriously slow—over 2700 times slower than the baseline test case—clearly indicting that MPI and the shared memory scheme needed for this case do not mix well. All asynchronous versions converged with about 1/3 the iterations than their synchronous counterparts, but any gains in computational performance were ultimately masked by additional communication overhead.

Specifically, performance speed-up for the 6 parallel implementations tested were as follows, with the non-optimized serial version normalized to one.

<i>Method</i>	<i>Speed-Up</i> ¹	<i>Config</i> ²	<i>Comment</i>
Sync/Serial	1	-O0	No Optimization
Sync/Serial	1.22	-O3	Best Intel C Optimization
Sync/pthreads	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/pthreads	1.45	2 threads	Errors for thread counts > 2
Async/MPI	0.0003	4 cpus	2728 times longer than baseline (7m. 22s.)
Async/openmp	0.40	2 threads	8 loops to converge

Introduction and Background

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 if \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 as follows.³

Specially, given that

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

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 simply the remainder of \mathbf{A} once \mathbf{D} is removed:

¹ Represent best case for specific implementation.

² Relevant variable setting for best case time

³ Special type-set images shamelessly copied from Wikipedia article on the Jacobi Method.

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

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

$$(\mathbf{D} + \mathbf{R})\mathbf{x} = \mathbf{b} \quad \text{and by distributing } \mathbf{x}, \text{ we get}$$

$$\mathbf{D}\mathbf{x} + \mathbf{R}\mathbf{x} = \mathbf{b} \quad \text{and by subtracting } \mathbf{R}\mathbf{x} \text{ from both sides, we get}$$

$$\mathbf{D}\mathbf{x} = \mathbf{b} - \mathbf{R}\mathbf{x} \quad \text{and by multiplying both sides by } \mathbf{D}^{-1}, \text{ we get}$$

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

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:

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{b} - \mathbf{R}\mathbf{x}^{(k)}).$$

We recall that with original system of linear equations, solving for \mathbf{x} , where

$$\mathbf{x} = \mathbf{A}^{-1}(\mathbf{b})$$

requires inverting the matrix \mathbf{A} , a computationally intensive operation, and hence, something we want to avoid.

It is here that one can see the genius/simplicity of the Jacobi iteration process. 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} , it is a simple matter to invert \mathbf{D} , a diagonal matrix, i.e.

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

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

And the rest of the operations are relatively straight forward and not overly computationally intensive.

Hence, we can write the element-based formula for the Jacobi iteration as:

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

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. Indeed, numerous technical papers indicate that 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. Succinctly put, a matrix \mathbf{A} is diagonally dominant if and only if, for every row i and column j :

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|.$$

And in that case, the Jacobi method is guaranteed to converge.

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 dominate matrix⁴.

- One option may use a QR iteration scheme. Normally used to simultaneously find all eigenvalues of a given $n \times n$ array, the process essentially increases the diagonal values in relations to all other value on the row, iterating until one essentially has a diagonal matrix whose elements are the sought after eigenvalues. Here, the same process could be used, but with an earlier stopping criteria that spots diagonal dominance, requiring less iterations than a full QR scheme.
- Ultimately, a future exercise could be to examine the performance tradeoff between using preconditioning and the Jacobi iterative process vs. the Gaussian elimination and its many variations on the theme.

Finally, as with any iterative process, the solution is arbitrarily achieved when the process has converged to some point whereby any additional iteration does not change the ‘answer’ more than some predetermined epsilon. For the sake of this exercise, I choose as a stopping criterion to use the Euclidean norm of the \mathbf{x} array under consideration.

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

and to then stop the process when the difference between the Euclidean norm for the i^{th} iteration and the Euclidean norm of $i^{\text{th}-1}$ iteration was less than a predefined parameter called TOLERANCE.

Specifically, the iterative process halts when

$$\text{TOLERANCE} < \text{abs}(\|\mathbf{x}_i\| - \|\mathbf{x}_{i-1}\|)$$

The JI and Synchronous vs. Non-synchronous Implementations

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. Indeed, one of the main characteristics of the Jacobi iteration is that it requires two arrays, say called \mathbf{x}_{old} and \mathbf{x}_{new} , to store the on-going updating of the \mathbf{x} array within each time step.

- 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. One version of JI—the Gauss-Seidel method—attempts to use some of the most recently updated version of the \mathbf{x}_i elements within a single iteration, but it operates within very strict confines.

However, because this project examined parallel implementations of the Jacobi iteration algorithm, it offered ample opportunity to also implement a number of so-called asynchronous versions that always use the most recent calculation for any given \mathbf{x}_i regardless of what time step it was produced.

⁴ One example is ‘Preconditioned Diagonally Dominant Property For Linear Systems with H-Matrices’ by Xue Zhong Wang et al. *Applied mathematics E-Notes* 6(2006).

Implementation and Test Results

Ultimately, seven version of the Jacobi iteration algorithm were implemented in C, compiled on the Intel C compiler, and tested on GMU's cds04.gmu.edu system. They were:

- 1) Synchronous serial (both baseline and optimized by the Intel C compiler)
- 2) Synchronous using pthreads
- 3) Synchronous using MPI
- 4) Synchronous using openmp
- 5) Asynchronous using pthreads
- 6) Asynchronous using MPI
- 7) Asynchronous using openmp

To test these programs, I needed to generate test data for \mathbf{A} and \mathbf{b} , in order to solve for \mathbf{x} . Using MATLAB, I constructed a 512 x 512 diagonally dominant matrix \mathbf{A} . Then I created a vector \mathbf{x} , which consisted of 512 elements, all equal to one.

Matlab was used to then solve for \mathbf{b} , by

$$\mathbf{Ax} = \mathbf{b}$$

which gave me a \mathbf{b} vector for input to the Jacobi iteration test.

The reason for doing it this way was that I could easily verify program correctness simply by looking at the calculated versions of \mathbf{x} , as I knew that the proper solution would always be $x = [1,1,1,\dots,1]$

In addition, in this case, the Euclidean norm of a the solution would be

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

and with every \mathbf{x} element =1, a proper solution would equal the square root of the number of elements in the array, which for my test cases is $\sqrt{512} = 22.627$

Time, space, and energy limitations prohibit write-ups on the specific issues surrounding each of these codes, but a summary/highlights of each, perhaps, will suffice.

Synchronous Serial: This codes best illustrates the heart of the JI that is central to all programs under consideration , and 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 to keeps the code from running away if there is no convergence .

Results:

For A size = 512 x 512, seeking $\mathbf{x} = [1, 1, \dots, 1]$,

- Un-optimized Version

On cds04.gmu.edu

- `icc -O0 serial_project_sorensen.c`
- `time ./a.out`

Real Time 0.162s

User Time 0.152s

Number of Loops to converge with tolerance of 0.000001= 18

Final Convergence Euclidean Norm = 22.627.

- Optimized Version

- `icc -O3 serial_project_sorensen.c`
- `time ./a.out`

Real Time 0.132s

User Time 0.108s

Number of Loops to converge with tolerance of 0.000001= 18

Final Convergence Euclidean Norm = 22.627.

Synchronous using pthreads: Here, the code dispatched threads for the specific loops seen above. The main issue here is that after each iteration, the program had to come together to calculate the new Euclidean norm to see if the \mathbf{J} has reached convergence. That turns out to be an interesting issue: threads cannot continue operation for any iteration without checking first to see if a solution has been reached. Consequently, they can only perform at most, one iteration before synchronizing to ascertain if the program is done.

Results:

For A size = 512x512, seeking $\mathbf{x} = [1, 1, \dots, 1]$

On cds04.gmu.edu

- `icc -pthread threaded_project_sorensen.c`
- `time ./a.out`

Synchronous Using Pthreads

Number of Threads	Real times (secs.)	User Time (secs.)	E. Norm	Convergence
1	0.137	0.124	22.627	18
2	0.119	0.100	22.627	18
4	0.107	0.096	22.627	18
8	0.105	0.092	22.627	18
16	0.124	0.100	22.627	18
32	0.150	0.132	22.627	18

Synchronous using MPI Here, MPI calls send off various chunks of calculations in parallel much like was done in threads, but now the data dispatches are sent across a (*much*) slower interprocessor network to other cpus. Since every mpi process needs the entire ‘old’ **x** array to perform its chunk of the iterative calculation, the **x** array had to be broadcast to every MPI process at every time step . Likewise, at each iteration, the root process had to broadcast the current Euclidean norm value so that each MPI process knows if it was finished running. The root processor was solely responsible for doing the Euclidean norm and associated stopping calculation

Results:

For **A** = 512x512, seeking **x** =[1, 1, ...,1]

On cds04.gmu.edu

using hostfile :

```
-----
cds03.gmu.edu
cds04.gmu.edu
cds05.gmu.edu
cds06.gmu.edu
-----
```

- eval `ssh-agent`
- ssh-add
- mpicc mpi_project_sorensen.c
- time mpirun -hostfile hostfile -np x ./a.out

Synchronous Using MPI

Number of MPI Machines	Real times (secs.)	User Time (secs.)	E. Norm	Max Iteration to Converge
1	0.845	0.028	22.627	18
2	0.608	0.040	22.627	18
4	1.887	0.044	22.627	18
8	5.807	0.024	22.627	18
16	6.543	0.056	22.627	18

Synchronous using openmp Here, it should be noted that it took approximately *90 seconds* to convert the serial code to parallel openmp using the single inserted line,

```
#pragma omp parallel for private(alpha,i,j)
```

illustrating, if nothing else, the ease of using openmp. In this case, the code was parallelized over the individual row summation calculation, and openmp handled the issues of dividing that task up among the available parallel threads. e

Results:

For **A** = 512 x 512, seeking **x** =[1, 1, ...,1]

On cds04.gmu.edu

- `icc -openmp openmp_project_sorensen.c`
- `time ./a.out`

Synchronous Using Openmp

Number of Threads	Real times (secs.)	User Time (secs.)	E. Norm	Max Iteration to Converge
1	0.335	0.128	22.627	18
2	0.300	0.136	22.627	18
4	0.318	0.166	22.627	18
8	0.325	0.184	22.627	18
16	0.320	0.120	22.627	18
32	0.319	0.166	22.627	18

Asynchronous using pthreads Here—in the asynchronous codes—is where things start to get funky. In essence, we needed to rewrite the codes so that each time a new element of x was calculated it became available to every other thread immediately. For pthreads, this was accomplished by doing away with the x_old and x_new arrays (a saving of memory space) and making x globally accessible.

An interesting effect emerged from this test almost immediately. The algorithm would not generate proper results in cases where threads did not give up control within relatively small increments on time. That is, in cases where one thread ran to completion before any other had begun, the answer was wrong.

- It seems that for this—and all asynchronous implementation to work—threads must give up control to other threads early and often. Indeed, it was necessary to ensure that within each iteration every thread performed the exact opposite of a thread-safe loop, surrendering program control to another thread.

There seems to be very little technical information available on how one can get a thread to operate in a ‘thread-risky’ environment, that is, allowing interruption at some predictable or even random time. For my purposes, I found that by inserting a single line into the main loop

```
for(w=0;w<WASTE;w++) { }
```

with `WASTE=1`,

it added no increase in execution time, but was enough to trigger a thread to yield program control to another thread. This is an interesting topic for further discussion.

Results:

For $A = 512 \times 512$, seeking $x = [1, 1, \dots, 1]$

On cds04.gmu.edu

- `icc -pthread async_threaded_project_sorensen.c`
- `time ./a.out`

Asynchronous Using Pthreads

Number of Threads	Real times (secs.)	User Time (secs.)	E. Norm ⁵	Max Iteration to Converge
1	0.142	0.128	22.267	7
2	0.111	0.096	22.627	8
4	0.121	0.116	24.50	7
8	0.118	0.111	25.086	6
16	0.137	0.128	25.444	6
32	0.136	0.120	25.702	4

Asynchronous using MPI Implementing an async version of MPI was non-trivial, but possible. Because we need to have the latest version all elements of \mathbf{x} read/write enabled to all threads at all times, I used the ‘shared memory’ capability of MPI, using remote memory access MPI_PUT and MPI_GET to transfer parts of the \mathbf{x} array from its ‘central location’ in the root process to the various other processors on an as needed basis.

- As such, these PUTS and GETs were the only section of the async code that could not be interrupted; hence, they were flanked by MPI_WIN_Fence statements.

Results:

For $\mathbf{A} = 512 \times 512$, seeking $\mathbf{x} = [1, 1, \dots, 1]$

Using cds04.gmu.edu

using hostfile :

```
-----
cds03.gmu.edu
cds04.gmu.edu
cds05.gmu.edu
cds06.gmu.edu
-----
```

- eval `ssh-agent`
- ssh-add
- mpicc async_mpi_project_sorensen.c
- time mpirun -hostfile hostfile -np x ./a.out

Asynchronous Using MPI

Number of MPI Machines	Real times (secs.)	User Time (secs.) ⁶	E. Norm	Max Iteration to Converge
4	757.0!!	756.8	22.627	7
8	442.0	441.2	22.627	8

⁵ Note the increasing error in the final solution.

⁶ Total CPU time

Here, it is pretty clear that we are getting the correct results, but with huge execution times. To verify these results, I generated a test set of data for 32 linear equations, which gave more manageable execution times, allowing for some analysis of MPI characteristics. Note that here, the expected Euclidean norm is $\sqrt{32} = 5.656$

For $A = 32 \times 32$, seeking $x = [1, 1, \dots, 1]$
 Using cds04.gmu.edu

Asynchronous Using MPI on a 32 x 32 Matrix

Number of MPI Machines	Real times (secs.)	User Time (secs.)	E. Norm	Max Iteration to Converge
1	0.333	0.010	5.656	7
2	6.670	6.270	5.656	7
4	11.982	11.19	5.656	8
8	27.166	24.12	5.656	8
16	5.786	1.71	5.656	10
32	9.050	1.37	5.656	16

Asynchronous using openmp Finally, we once again enjoyed the ease of openmp, where it took minimal time—perhaps, three minutes—to figure out a way to async the earlier version. Simply put, all that was required was combining the x_old and x_new array into one single x_new . Then the previous openmp directive was changed to reflect that the x_new array was common to all :

```
#pragma omp parallel for private(alpha,i, j,) shared(x_new)
```

Results:

For $A = 512 \times 512$, seeking $x = [1, 1, \dots, 1]$

On cds04.gmu.edu and

- set OPM_NUM_THREADS = [1,2,4,8,16,32]
- export OPM_NUM_THREADS
- icc -openmp async_openmp_project_sorensen.c
- time ./a.out

Asynchronous Using Openmp

Number of Threads	Real times (secs.)	User Time (secs.)	E. Norm	Max Iteration to Converge
1	0.311	0.176	22.627	9
2	0.299	0.088	22.627	8
4	0.346	0.136	22.627	7

8	0.314	0.108	22.627	8
16	0.318	0.116	22.627	8
32	0.315	0.104	22.627	8

Data Analysis and Summary:

In the best case, a synchronous pthread version realized a 1.54 speed up over the baseline serial case. The slowest implementation, the asynchronous MPI version, ran gloriously slow—over 2700 times slower than the baseline test case—clearly indicting that MPI and the shared memory scheme needed for this case do not mix well. All asynchronous versions converged with less iterations than their synchronous counterparts, typically needing only about ½ - 1/3 the iterations, but any gains in computational performance were ultimately masked by additional communication overhead. Specifically,

<i>Method</i>	<i>Real Time</i> ⁷	<i>Config</i> ⁸	<i>Comment</i>
Sync/Serial	0.162	-O0	No Opt.
Sync/Serial	0.132	-O3	Best Opt. inc. loops
Sync/pthreads	0.105	4-8 threads	18 loops to converge
Sync/MPI	0.608	2 cpus	Degrades rapidly with cpu#
Sync/openmp	0.300	2 threads	Constant across thread #
Async/pthreads	0.111	2 threads	Errors for thread counts > 4
Async/MPI	442.0	8 cpus	Wow
Async/openmp	0.299	2 threads	8 loops to converge

There are a number of observations that can be drawn from these results. They are, in no particular order:

- For the asynchronous vs. its synchronous counterpart, the async versions typically needed about one-third the iterations to converge to the solution with the same tolerance epsilon, which is cool. Chaos rules.
- In almost all cases—synchronous and asynchronous—the total user time to complete the iteration was essentially the same—ranging from 0.1 – 0.15 seconds, indicating that the variance in real time was due largely to communication and parallel overhead and not to any increase in computational activity.
- 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—both which I suspect were running the same chip.
- Asynchronous MPI failed in a fairly spectacular manner, clearly illustrating the overhead involved with MPI data transfer rates between processors needed to implement a global shared memory capability. Indeed, it was over 2,700 times slower than the base line code.

⁷ For this table, the real time listed shows the best in class (BIC) performance of that code within its range of variable parameters, i.e. # of threads, number of cpus. Etc.

⁸ Relevant variable setting of BIC time

- Asynchronous pthreads present an interesting case. For one and two threads, the process iterated to the right answer. But for increased thread counts, the process iterated to a stable, albeit wrong solution—that is, forcing iteration counts up five or six orders of magnitude did not drive the process to a correct solution. In addition, the error of the erroneous Euclidean norms seems to increase linearly with the increase in thread count. I am not entirely sure why this would happen, but I suspect it has to do, once again, with the inability to dictate when a particular thread ‘relinquishes’ control in the course of a loop.

And finally; in two cases—synchronous and asynchronous threads—the performance gains were more than 50 percent of the non-optimized baseline. This is reasonable considering that we are only dealing with 2 cpu systems and the maximum potential speed up is limited to 2x.

- But perhaps more important, both threaded versions outperformed the optimized serial code that incorporated the best tricks that the Intel C compiler had to offer—even on a highly parallelizable code such as the Jacobi iteration. Considering that Intel spends \$3-4 billion on R&D every year—and I don’t—that’s not too shabby.