CS 521 Parallel Algorithms and Architectures
Department of Computer Science
University of British Columbia

Solving the Poisson Problem in Parallel with S.O.R.

Landon Boyd

blandon@cs.ubc.ca
Department of Computer Science
University of British Columbia
Vancouver, BC, Canada

Abstract

The Poisson Problem, $\nabla \cdot \nabla x = b$, is a sparse linear system of equations that arises, for example, in scientific computing. For this project, I describe a parallel Successive Over-Relaxation (SOR) algorithm for solving the Poisson problem and implement it in a C library using Message Passing Interface (MPI). I evaluate the performance of my implementation on a single multicore machine and in a cluster.

Keywords: parallel computing, SOR

1 Introduction

In the realm of scientific computing, one often wishes to solve a linear system of equations, $A\mathbf{x} = \mathbf{b}$, where A is sparse, meaning that it has many zero entries. In particular, this arises when solving partial differential equations on a grid, as when simulating fluids for example [1].

The *indirect* class of sparse linear solvers approach the solution iteratively until a specified error tolerance is met. Historically, the Jacobi and Gauss-Seidel methods have been used, with an additional speed-up in convergence obtained by way of *Successive Over-Relaxation* (SOR) [2]. SOR enhances the aforementioned methods by amplifying the step that the method would have taken on a particular iteration. The amount of amplification is determined by a relaxation parameter.

Whereas Jacobi is trivial to implement in parallel, Gauss-Seidel is more challenging because computing element x_i for an iteration may use new values of other elements x_j ($j \neq i$) in $\mathbf{x}$, previously computed in the same iteration. To cope with this, a colouring scheme can be used to assign processors to elements that can be processed independently [3, 4].

For example, when solving the Poisson problem $\nabla \cdot \nabla x = b$ on a grid, each grid node is dependent on its left, right, top and bottom neighbors. Two colours (say, red and black) are sufficient to ensure that each node is a different colour than all four neighboring nodes in its stencil. Then, the Gauss-Seidel step can proceed in two phases, updating the red nodes in parallel first and then the black nodes in parallel.

For this project, I have implemented a C library using MPI that takes a sparse $n \times n$ matrix, A, an $n \times 1$ right-hand-side, $\mathbf{b}$, and relaxation parameter, r, and solves the linear system $A\mathbf{x} = \mathbf{b}$ using SOR in parallel.

Because the colouring is highly dependent on the problem being solved, a general sparse solver might also take this colouring scheme as input or attempt to determine it from the structure of the sparse matrix. For the sake of simplicity and efficiency, my library only handles systems of the form dictated by the Poisson problem, ie. $A = \nabla \cdot \nabla$, where $\nabla \cdot \nabla$ is the finite difference matrix used to calculate the Laplacian of a quantity on a simulation grid.

For example, a 2-D simulation grid of size $p \times q$ will have a corresponding A of size $pq \times pq$. The quantity at grid point (i, j) will have a corresponding row with five non-zero entries corresponding to the standard 5-point stencil. This extends to 3-D with a 7-point stencil.

2 Strategy

Since the Poisson problem arises when simulating physical processes on a regular grid over space, the natural topology for parallelization is also a grid, where each processor handles a subset of the simulation grid. Ideally, the bulk of an application (not just the Poisson solve) would be parallelized using this strategy.

For example, in a fluid simulation, in addition to the Poisson solve for pressures, there are steps to advect velocities through the grid, update a liquid surface representation, etc. Many of these steps are also parallelizable. Rather than a master process doling out portions of each of these tasks to workers, it would be preferable for each processor to "own" a portion of space in order to minimize communications.

My library, *MpPoisson*, accommodates the case of a sequential application that wants to use a parallel Poisson solve (master-worker), and also when the Poisson solve is part of a fully parallel application.

2.1 Application Programming Interface

The first routine called by a user, `get_context(dims[],ndims,comm,maxp)`, takes as input *dims*, the dimensions of the simulation domain, *ndims*, the dimensionality of the problem (2 or 3), *comm*, a communicator, and *maxp*, the maximum number of processes to assign to the grid. It returns a Context structure, including:

- an MPI communicator
- the range of simulation cells owned by this process
- a reference to the data area for this node

All subsequent API calls take this context as input.

Ideally, *maxp* will be a perfect square for a 2-D problem or cube for 3-D, allowing the domain to be split into equal sized squares or cubes. Recognizing that this is not always the case, `get_context()` looks for factors of *maxp* that will partition the space into rectangular subdomains with roughly minimal surface area, by the following procedure:

1. Let there be one subdomain consisting of the entire domain;
2. Find the prime factors of *maxp*, in descending order: $f_1 \dots f_k$;
3. For each factor f_i in $f_1 \dots f_k$
   - Split each subdomain into f_i parts along its longest dimension;

If the subdomains are too skinny, the benefit of the additional processors may be nullified by the overhead of communication between them (see section 4.1). As a future enhancement, `get_context()` could evaluate the "niceness" of the partitions and use fewer processors than *maxp* when appropriate.

In the master-worker case, worker processes then call `worker(context)`, which waits for messages from the master. From this point until the master closes its context, the worker processes are completely in the control of the library.

The next step is to specify the coefficients in the large, sparse A matrix on the left side of the Poisson equation. Each simulation grid cell corresponds to a row with up to 5 (2-D) or 7 (3-D) non-zero coefficients. In a fluid simulation with liquid surfaces and/or moving solids, these coefficients may change over time.

The `set_coefficients(context,vals)` command tells the library where to find the coefficient data. This data is owned by the user. The user must ensure that this pointer is valid for the lifetime of the context, and the user is responsible for deallocating the associated memory when finished. In the master-worker model, `set_coefficients(context,vals)` also sends the coefficients to the worker processes.

The Poisson equation also needs values for the right hand side. These can be set similarly to the coefficients using `set_rhs(context,vals)`.

Once the system has been set up, `solve(context,r)` iteratively solves it using relaxation parameter r. In the master-worker model, only the master calls this function; otherwise all processes must call it.

`solve()` returns the number of iterations used to find the solution, or zero if a solution was not found. `get_solution(context)` returns a pointer to the solution data in the context. In the master-worker model, the master context will have all of the results; otherwise each processor will have meaningful values only in the entries for cells owned by that processor.

2.2 Parallel S.O.R. Solve

To solve a system $A\mathbf{x} = \mathbf{b}$, Gauss-Seidel algorithm visits each entry of $\mathbf{x}$ in order. It updates the ith entry as follows:

$$x_i^{new} = \frac{f_i - a_{i,1}x_1^{new} - \cdots - a_{i,i-1}x_{i-1}^{new}}{a_{i,i}} - \frac{a_{i,i+1}x_{i+1}^{old} - \cdots - a_{i,n}x_n^{old}}{a_{i,i}}$$

Where the "new" values are from the current iteration, and "old" values are from the previous iteration. It does this repeatedly until the solution converges. SOR accelerates this convergence by using the relaxation parameter, r, as follows:

$$x_i^{new} = x_i^{old} + r\Big(\frac{f_i - a_{i,1}x_1^{new} - \cdots - a_{i,i-1}x_{i-1}^{new}}{a_{i,i}} - \frac{a_{i,i+1}x_{i+1}^{old} - \cdots - a_{i,n}x_n^{old}}{a_{i,i}} - x_i^{old}\Big)$$

Since it uses values x_j^{new}, $j < i$ from the current iteration, this complicates parallelization, but we can take advantage of the sparse structure of the matrix. Consider the simulation grid and corresponding A matrix and $\mathbf{x}$ vector shown in figure 1. The fifth row in A, corresponding to element x_5 in simulation cell $(2, 2)$, has 5 non-zero entries; two of which, $a_{5,2}$ and $a_{5,4}$ are

left of the diagonal. Therefore, calculating x_5^{new} will require x_2^{new} and x_4^{new}.

If we reorder the rows as shown in figure 2, now the simulation cell $(2, 2)$ corresponds to x_3. The third row in A is has no entries left of the diagonal, therefore it is not dependent on any x^{new} values. This is true of all of the red coloured entries in the top half of A. Conversely, the black entries in the bottom half of A are dependent only on the red entries.

To solve this system in parallel, we can update all the red entries first in parallel, and then all the black entries in parallel. That is the heart of the parallel SOR algorithm. Starting from an initial guess of $\mathbf{x} = 0$, each processor does the following for each iteration:

1. calculates all its red entries
2. exchanges boundary red values with adjacent processors
3. calculates all its black entries
4. exchanges boundary black values with adjacent processors

x_7	x_8	x_9
x_4	x_5	x_6
x_1	x_2	x_3

$$\begin{pmatrix} -4 & 1 & & 1 & & & & & \\ 1 & -4 & 1 & & 1 & & & & \\ & 1 & -4 & & & 1 & & & \\ 1 & & & -4 & 1 & & 1 & & \\ & 1 & & 1 & -4 & 1 & & 1 & \\ & & 1 & & 1 & -4 & & & 1 \\ & & & 1 & & & -4 & 1 & \\ & & & & 1 & & 1 & -4 & 1 \\ & & & & & 1 & & 1 & -4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix}$$

Figure 1: sequential node assignments

x_4	x_9	x_5
x_7	x_3	x_8
x_1	x_6	x_2

$$\left(\begin{array}{cccccc|cccc} -4 & & & & & & 1 & 1 & & \\ & -4 & & & & & 1 & & 1 & \\ & & -4 & & & & 1 & 1 & 1 & 1 \\ & & & -4 & & & & 1 & & 1 \\ & & & & -4 & & & & 1 & 1 \\ \hline 1 & 1 & 1 & & & & -4 & & & \\ 1 & & & 1 & 1 & & & -4 & & \\ & 1 & 1 & & & 1 & & & -4 & \\ & & 1 & 1 & 1 & & & & & -4 \end{array}\right) \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{pmatrix}$$

Figure 2: re-ordered node assignments

After each iteration, each processor broadcasts a message containing the sum of squares of the residuals, $\sum_i [b_i - (Ax)_i]^2$, for its elements. Each processor sums its own value with those received from other processors to obtain the squared L_2 norm of the overall residual. If it is below a certain threshold, execution terminates successfully. If the threshold hasn't been met and a maximum iteration count is exceeded, execution terminates unsuccessfully. Otherwise, another iteration occurs.

3 Details of the Implementation

To represent a sparse matrix, I use a `SparseMatrix` structure containing the following arrays:

- `row_start`: for each row, the first index into `value`; a trailing entry with the count of values;
- `value`: the non-zero values in row order;
- `column_index`: for every value, the index of its column;

For an $n \times n$ matrix with m non-zero values, the size will be $n + m$ ints plus m doubles. See figure 3 for the representation of the first two rows of the matrix in figure 1.

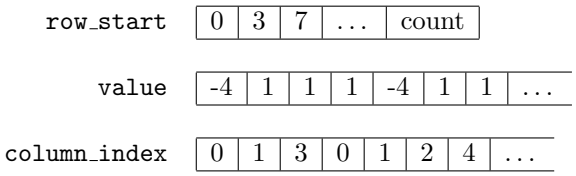

Figure 3: `SparseMatrix` example

3.1 `get_context()`

In `get_context()`, `MPI_Cart_Create()` is used to create a Cartesian MPI topology using the output of the partitioning algorithm described in section 2.

A custom MPI error handler is defined using `MPI_Errhandler_set()`. In the case of a fatal MPI error, detailed information is output to the console before the process exits, as opposed to the default MPI error handler which exits without any output.

Each context is populated with an array of `Neighbor` structs, for each subdomain adjacent to the processor. These structs contain a range of cells on the boundary and a vector pointing into the neighbor. This is used during the solve when neighboring black or red values are exchanged between neighbors. There are also send and receive buffers for the neighbors. These are allocated ahead of time to avoid work during the Gauss Seidel iterations.

In the case of a master process, the context also is populated with an array of `Worker` structs for each of the worker processors. The master needs to know the range of cells owned by each worker in order to send it the appropriate data in `set_coefficients()` and `set_rhs()`, and to gather the solution in `solve()`.

3.2 `worker()`

The first thing `worker()` does is allocate a bunch of memory for the coefficients and right hand side which will be received from the master. Again, as much memory allocation is done up-front so that this doesn't have to be done during individual solves.

The worker process than proceeds in a loop driven entirely by messages from the master. `MPI_Irecv()`s are issued for the following message tags:

`MPP_MSGID_COEFFS_ROWSTART` : `row_start` vector for coefficients from the master

`MPP_MSGID_COEFFS_VALUE` : `value` vector for coefficients from the master

`MPP_MSGID_COEFFS_COLINDEX` : `column_index` vector for coefficients from the master

`MPP_MSGID_RHS` : `rhs` vector from the master

`MPP_MSGID_SOLVE` : signal from the master to do a solve

`MPP_MSGID_STOP` : signal from the master to quit

The worker blocks in `MPI_Waitany()` until one of these is received, takes the appropriate action, and then repeats the loop until an `MPP_MSGID_STOP` request is received as the master's context is closed.

3.3 `set_coefficients()`

Whenever data is sent between processors, only the values themselves are sent; for example, in `set_coefficients()`, it is not necessary to specify the row containing each value because they are sent in a prescribed order. The sender and receiver loop over the dimensions of the appropriate subdomain in the same order when sending and receiving.

Sending a subset of a sparse matrix is done as follows. Suppose a matrix contains 9 rows, of which rows 1, 2, 5 and 6 are to be sent (figure 4). The appropriate values (and column indices, not shown) are packed into a send buffer, `valbuf`. Another buffer, `rsbuf` is populated with the four indices in `valbuf` of the first values of each row. The last entry in `rsbuf` is the size of `valbuf` (figure 5).

A corresponding unpacking procedure is done on the receiving end.

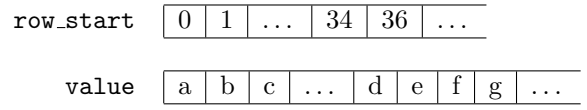

Figure 4: `SparseMatrix` subset

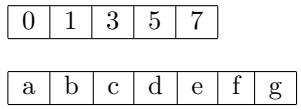

Figure 5: packed `SparseMatrix` subset

3.4 `solve()`

`solve()` implements the parallel SOR algorithm described in section 2.2. There are steps to exchange "red" or "black" intermediate solutions with neighboring processors. This is done using point-to-point communication (`MPI_Isend()` and `MPI_Irecv()`. It might have also been possible to use `MPI_Sendrecv()` but the nonblocking calls seemed like the easiest way to avoid deadlock. Each processor posts its receives and then initiates its sends.

An `MPI_Allreduce()` operation is used to sum the residuals of all subdomains to test for convergence. During testing, I found that performance was improved significantly by performing this step only after every 10 SOR iterations [5].

Finally, in the master-worker model, when a solve has completed, the results must be sent to the master. This is done using `MPI_Allgatherv()`. The workers' results are concatenated onto a receive buffer for the master, which unpacks them into the appropriate locations in the solution vector.

4 Performance

4.1 Efficiency

A series of tests were performed to evaluate the efficiency of the parallel SOR implementation. The test data consisted of a standard 7-point Laplacian coefficient matrix A corresponding to a 3-D domain with a Dirichlet boundary condition of zero around the boundaries. The right hand side $\mathbf{b}$ was generated by multiplying A by a random vector $\mathbf{x}$. The solution derived by SOR was then compared to $\mathbf{x}$ to verify correctness.

After some experimentation, I settled a relaxation parameter of 1.97. This seemed to work the most consistently, allowing domains of various sizes to be solved in less than 1000 iterations. It should be noted that this is by no means the optimal value for many domain sizes but that was acceptable since my goal was to evaluate the parallel implementation, not SOR.

These tests were performed using the peer model, ie. each processor had the input data before the test

started and did not send the results to a master.

Domains of various sizes ($n \times n \times n$) were tested on a single machine with 8 AMD Opteron 8220 cores at 1 GHz and 128 gb of RAM. Various numbers of processors p were used. Each test was run three times and the best result used. The times to perform the solves are shown in table 1.

p	$n = 40$	$n = 100$	$n = 180$
1	4.72	81.69	614.57
2	5.03	61.23	602.52
3	4.19	52.58	487.54
4	2.58	33.52	165.08
5	3.26	38.54	319.6
6	1.84	24.38	137.7
7	2.93	33.69	256.15
8	0.90	16.96	98.07

Table 1: Solve times on single 8-core machine

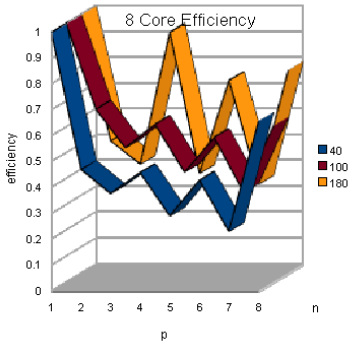

Figure 6: efficiency on single 8-core machine

The resulting efficiency is plotted in figure 6. Note that prime numbers of processors resulted in reduced efficiency as the subdomains had large surface areas resulting in increased communication costs. Efficiency improved as the problem sizes got bigger, as expected since a bigger domain means more local work compared to communication as the volume of the subdomains increases faster than the surface area between them.

In subsequent tests, I tried using more processors than cores. For example, on the 8-core machine, I tried using 16, 27 and 64 processors for a 200^3 domain, but these all resulted in slower performance than using 8 processors. On a dual-core machine there appeared to be little difference from using more processors than cores.

Similar tests were performed on four 100Mbps-networked dual core machines with the following specifications:

- Intel Pentium 4, 3.2 GHz, 4 gb RAM
- Intel Core 2 Duo, 3.0 GHz, 3 gb RAM
- Intel Pentium 4, 3.0 GHz, 1 gb RAM
- Intel Pentium 4, 3.0 GHz, 1 gb RAM

Although as far as I could tell no one was using these machines at the time of the tests, it was not a totally controlled environment. The single processor baseline was obtained using the fastest machine so efficiency numbers were scewed downward. With that in mind, the results are in table 2 and efficiency in figure 7.

p	$n = 40$	$n = 100$	$n = 180$
1	4.04	66.39	417.26
2	4.91	54.09	295.09
3	4.89	48.20	240.61
4	3.26	30.60	150.84
5	4.89	46.39	245.51
6	3.70	36.02	208.79
7	5.76	51.82	502.53
8	2.40	24.06	145.52

Table 2: Solve times on four dual-core machines

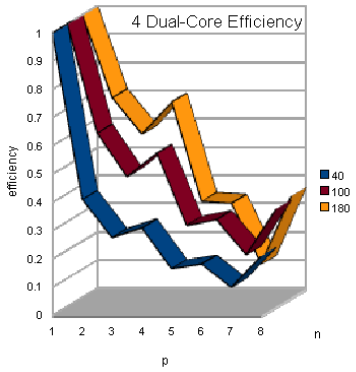

Figure 7: efficiency on four dual-core machines

Overall, the speedup was much less for the cluster compared to the 8 core machine, presumably due to communication costs over the 100 Mbps network.

4.2 Memory Usage

Testing domains of size 200^3 or larger was not practical for two reasons. One was that on larger domains, SOR failed to converge within 1000 iterations. Another was memory usage. `top` showed that at that problem size, the test process was consuming about 860 mb of memory. One of the test machines with 1 gb of physical memory started to slow down considerably at this point, presumably due to paging.

One way to address this would be to use a compressed coefficient matrix as per [6]. They use a table of unique stencils and each row contains a pointer into that table. In most fluid simulations, this would save a lot of space since all simulation cells surrounded

by equal density air, for example, would have identical stencils.

4.3 Master-Worker Overhead

The master-worker model adds the additional steps of sending input data and receiving solutions from workers. In tests on a single dual-core machine, using the master-worker model did not appear to hinder performance. In fact, it surprisingly performed slightly faster. However, when data was sent over the network, there was a detrimental effect on performance.

The amount of data sent in the master-worker model is proportional to the number of simulation cells. With a 100^3 domain on four machines, the additional time for the master-worker test was 5 seconds (5×10^{-6} seconds per cell). For a 150^3 domain, it was 18 seconds (also about 5×10^{-6} seconds per cell). When using one core per machine, this overhead amounted to about 15 percent of the total time.

This could perhaps be improved by using collective communication (e.g. `MPI_Scatterv`) instead sending to each worker synchronously in turn. This would require larger send buffers in the master. The amount of communication could also be reduced using the stencil table mentioned previously, and one might also consider only sending values that have changed since the previous solve.

5 Conclusions

The efficiency of this parallel SOR implementation varied depending on the size of the domain and the number of processors being used. Performance was better on a single multi-core machine than on a cluster due to the significant amount of communication necessitated by the algorithm.

Scalability is limited somewhat by the amount of memory and communication required for larger problems, though this could be addressed using a stencil table or other techniques.

Due to the difficulty of selecting an optimal relaxation parameter, and because it often takes many iterations to converge, SOR in itself is of limited use. However, this implementation could form a building block for a multigrid solver, one of the most effective ways to solve sparse linear systems [5, 6].

References

[1] R. Bridson. Fluid Simulation for Computer Graphics. *A K Peters*, 2008.

[2] R. Bridson. CPSC 542g: Scientific Computing Lectures. `http://people.cs.ubc.ca/~rbridson/courses/542g-fall-2008/lectures.html`, 2008.

[3] P.B. Hansen. *Studies in computational science: parallel programming paradigms.* Prentice Hall, 1995.

[4] L.M. Adams and H.F. Jordan. Is SOR color-blind? *SIAM Journal on Scientific and Statistical Computing*, 7:490, 1986.

[5] M. Šterk and R. Trobec. Parallel performance of a multigrid poisson solver. In *Proceedings of Second International Symposium on Parallel and Distributed Computing*, pages 238–243.

[6] M.B. Nielsen, B.B. Christensen, N.B. Zafar, D. Roble, and K. Museth. Guiding of smoke animations through variational coupling of simulations at different resolutions. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 217–226. ACM, 2009.