

Efficient Algorithms for Computing the *Krylov* Space involving *Poisson's* Equation

Andy Carle and Yuk Fai Tham
Prepared for CS267 with Professor James Demmel
Spring Semester, 2005

Background

Poisson's equation is the most common example of an elliptic partial differential equation (PDE). In three-dimensional space, it can be stated as

$$-\frac{\partial^2 u(x, y, z)}{\partial x^2} - \frac{\partial^2 u(x, y, z)}{\partial y^2} - \frac{\partial^2 u(x, y, z)}{\partial z^2} = f(x, y, z) \quad (1)$$

where $u(x, y, z)$ is the desired solution and $f(x, y, z)$ is the forcing function imposed on the domain. The one and two-dimensional form of *Poisson's* equation can be obtained from (1) by keeping the appropriate second-order spatial derivatives. This equation arises in many scientific applications, such as heat flow and fluid flow [1]. The problem domain can be divided into distinct data points, and the solution is approximated by the values at each data point. By using the appropriate stencil (figure 1), the second order derivatives can be approximated.

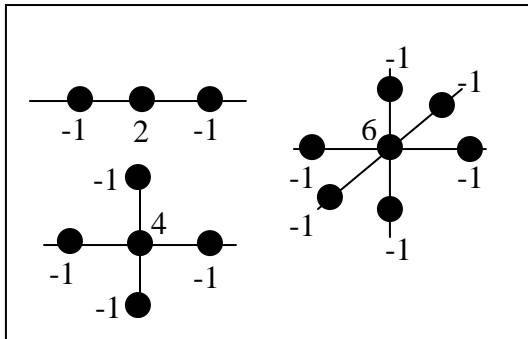


Figure 1. Stencils for *Poisson's* equation in 1-D, 2-D, and 3-D.

As a result of the discretization, (1) can be expressed as a linear system of equations in the form of $Ax=b$. The resulting square matrix A is sparse since the stencils only allow each data point to communicate with its nearest neighbors. The complexity of the matrix increases with increasing dimensions since the number of nearest neighbors increases as well. Figure 1 shows that matrix A for the 1-D *Poisson's* equation is a tri-diagonal matrix. Figures 2 and 3 show that the structure of the matrix increases in complexity as the pattern of the nearest neighbors with each data point changes throughout the solution domain.

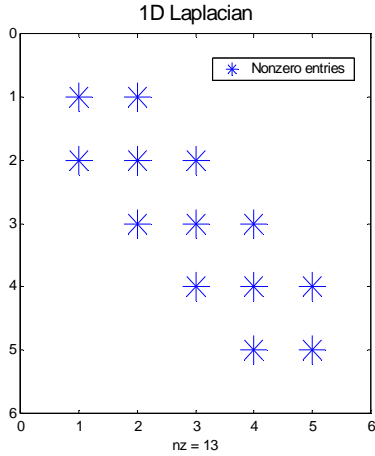


Figure 2. 1-D Laplacian matrix with N=5.

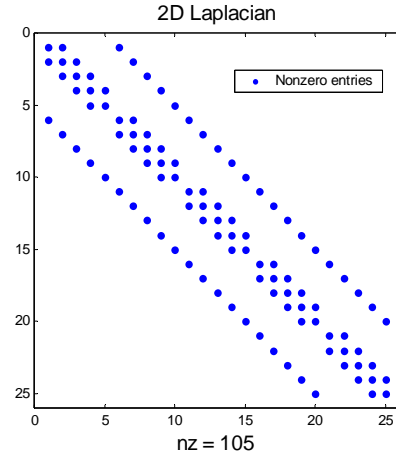


Figure 3. 2-D Laplacian matrix with N=5.

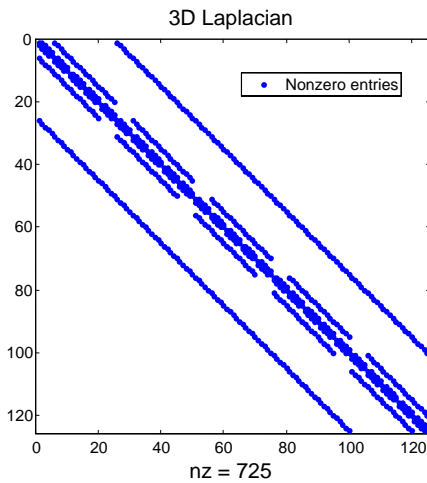


Figure 4. 3-D Laplacian matrix with N=5.

The *Krylov* space is defined as the following:

$$K(A,x) = \text{span} \{x, Ax, A^2x, A^3x, A^4x, \dots, A^kx\}$$

where the power k is the highest power desired. Vectors from the *Krylov* space are important because they are used in the *Lanczos* method for factorization or the Conjugate Gradient (CG) method in computing iterative solutions. Since the matrix A that results from the *Poisson's* equation is mostly sparse, the *Krylov* space that spans matrix A can be calculated efficiently. The worst possible way (which should never be done) is to compute all the powers from A to A^k and apply to the vector x . Not only will the subsequent matrices A^2 to A^k become dense, but matrix-matrix multiply involves more computations ($O(n^3)$). The efficient way is to *apply* A first to x , then to Ax , and so on. The method for the 1-D case is straightforward (figure 5), but the method for the 2-D case is more complicated (figure 6).

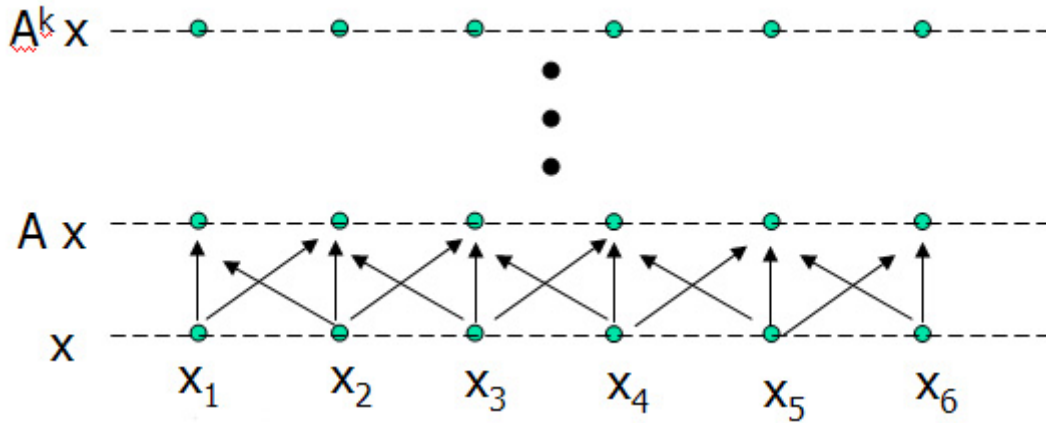


Figure 5. Forming the Krylov space of the 1-D Laplacian matrix.

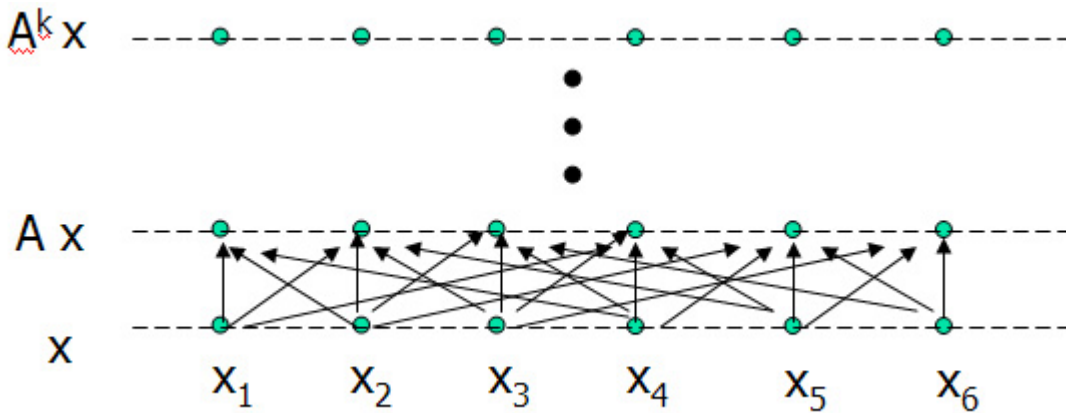


Figure 6. Forming the Krylov space of the 2-D Laplacian matrix.

Un-Optimized Method

The structures of the A matrices are different for the 1-D, 2-D, and the 3-D cases. To implement the straightforward method of calculating the *Krylov* space, simply utilize the structures of each A matrix and apply each A matrix to the *vector* x to form Ax . Similarly, perform the same task for the *vectors* $Ax, A^2x, A^3x, A^4x, A^5x, \dots$ etc. This method is substantially more efficient than calculating the necessary powers of A and then multiplying the vector x through each, as previously discussed.

Implementing this algorithm in parallel incurs a very high communication cost. The most clear way to split up the work among multiple processors is to divide the vector x among all processors and perform a piece of each calculation A^kx on the processor that holds the appropriate portion of the vector. While this is very straightforward, it can be seen that this implementation requires, in the 1-D case, each processor to perform a communication with each of its immediate neighbors to calculate each A^kx . The problem is even worse for the 2-D and

3-D cases, where the data that is needed to perform each processor's results is much more spread out among the other processors. The number of messages passed per processor is on the order of $2k$ in the 1-D case and pk for higher dimensions.

On a typical large-scale parallel machine this type of message passing is very expensive. Each message that is passed comes at the cost of a high latency penalty, the time that it takes for data sent from one processor to begin arriving at another. This latency penalty is typically orders of magnitude longer than the time taken to perform a floating point operation (flop) or actually transmit a data item. As the number of flops performed per message decreases so too will the performance of an algorithm in parallel, as the time spent waiting for messages to arrive begins to dominate the overall run time. Similarly, it is clear that passing a few very large messages is preferable to sending many small messages, as the time spent transmitting a data item is typically modest compared to the latency cost associated with the message as a whole. These insights motivate the use of a cleverer algorithm for calculating the *Krylov* space.

Optimized Method

Given the high cost of message passing among processors, it would appear preferable to organize the algorithm such that a processor can calculate all k powers of A^kx with just one set of messages. This can be accomplished by passing much larger pieces of the vector x to each processor and performing a (potentially large) number of extraneous calculations (See Figure 7). These calculations are extraneous because they are redundant to calculations taking place on other processors. However, given the very high flop rates of most parallel machines, this is a small price to pay to avoid extra messages. Most methods that require calculation of a *Krylov* subspace depend upon much more data (size of the matrix A and vector x) than powers of A (k). This leads to acceptable potential performance from the optimized algorithm (i.e. a situation as bad as what is showing in Figure 7 is unlikely to actually occur).

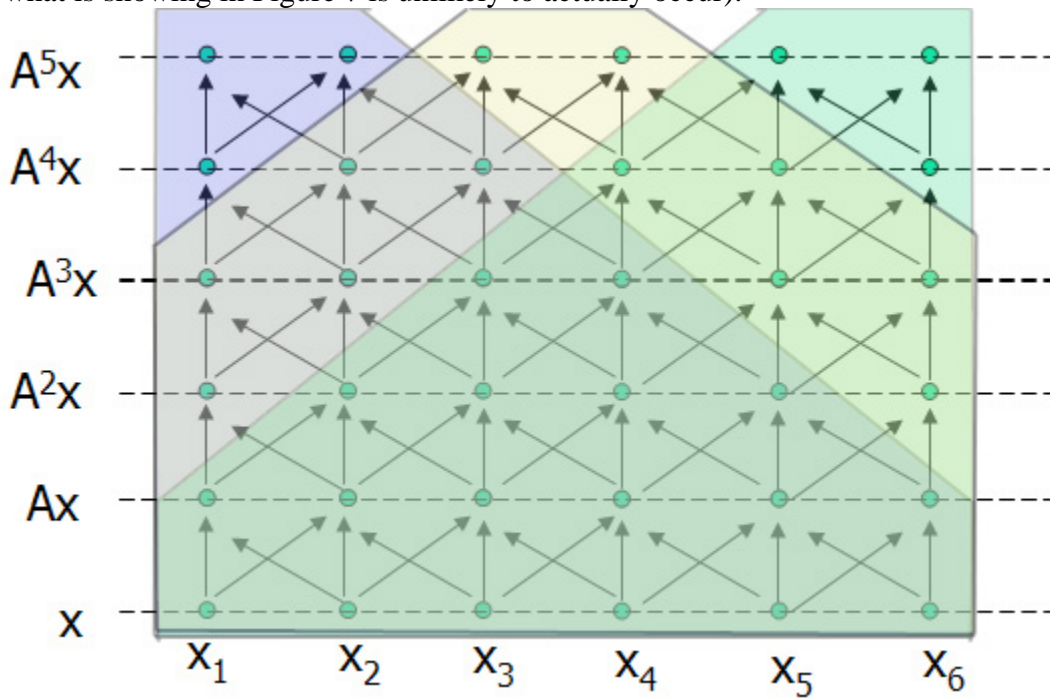


Figure 7 – Simple 1-D *Krylov* space showing redundant calculation between 3 processors

Predicted Results

Using a simplified model of the calculation of the *Krylov* subspace in parallel, it is possible to make predictions about the results of the optimized algorithm. Simplified (i.e. “order of...”) models of the computation and communication needs of the 1-D and 2-D algorithms can aid in these predictions. In the following notation, k represents the maximal power of matrix A that is to be found, n represents the length of the side of the mesh/stencil from which A is derived, and p is the number of processors the algorithm is run on.

In the 1-D case, the flops required per processor can be approximated relatively easily. Ignoring edge cases (those processors without two neighbors), each processor needs to process the $A^k x$ terms for all k and every entry in x that it owns. These $\frac{kn}{p}$ calculations are the same that would

be done in the un-optimized version of the algorithm. In addition to these calculations, each processor must also compute two extra numbers for level $k-1$, four extra for level $k-2$, etc... This leads to an additional $\sim k^2 - k$ calculations. So, a reasonable approximation for the flops per processor in the 1-D case is $k^2 - k + \frac{kn}{p}$. The number of messages sent (and, symmetrically,

received) by each processor in the 1-D case is also easily discovered. Each processor will need to communicate with its immediate neighbors, that neighbor's other neighbor, and so forth until all of the necessary entries in x have been gathered. The number of “horizontal” steps required to make this work is directly proportional to k , while the number of processors touched in this process is a function of $\frac{n}{p}$. Again ignoring edge cases, the total number of messages

sent/received by each processor in the 1-D case is $2 \left\lceil \frac{kp}{n} \right\rceil$. The number of words involved in

these messages can be similarly computed. Two double-precision floating point numbers must be sent to each processor per level in the calculation. Therefore, the number of words sent per process (again ignoring the edge cases) is about $8k$.

Similar techniques can be used to find reasonable expected values in the 2-D case. The flop rate for the 2-D case grows at a faster rate than in the 1-D case due to a recurrent dependence upon distant data elements. These force each processor to know about fairly large tree of data, potentially far away from its own data. The resulting number of flops is in the neighborhood of

$\frac{2}{3}k^3 + k^2 - 7/3k + 4/3 + \frac{2kn^2}{\sqrt{p}} - \frac{n^2}{\sqrt{p}}$ which would be dominated by the k^3 term in a generic

domain, but given our assumption that n is many orders of magnitude larger than k , is dominated in practice by the n^2 terms. It is possible to sort out exactly how many messages each processor will send/receive in the 2-D case (and higher orders), but the only important thing to realize is that it will be strictly $< p$, and not a function of k as would be the case in the un-optimized case. Similarly, there can be found a very rough upper bound on the number of words communicated

by each processor of $k^2 + 7k + \frac{4kn - 4n}{\sqrt{p}}$.

Implementation

As part of our work on this project we have implemented the 1-D and 2-D cases in Matlab (to ensure proper results) and C (using MPI for parallel communication). The sequential, non-optimized 1-D, 2-D, and 3-D cases and the optimized, parallel 1-D and 2-D versions of the code can be found at <http://www.cs.berkeley.edu/~acarle/cs267/project/>. Regrettably, this portion of the project was tackled very late. As such, both versions are buggy beyond being reliable enough to do any real analysis. The main goal with the implementation part of this project was to test for a match between our predicted (modeled) parameters and real world performance. Unfortunately, this work could not be finished with the current buggy implementation and an overall lack of time. However, there does appear to be a general correlation between our predications and the test cases that we can get to work in our implementation. In particular, we have confirmed that (at least on our test bed, the CITRIS cluster) n must be many orders of magnitude larger than p and k to achieve a reasonable parallel speedup. That is to say, if n is too small, adding additional processing power is not helpful. This makes sense, as further dividing the vector x results in more messages sent in the initial communications phase of the algorithm, required for large values of k .

We regret not being able to peruse this issue (and squash our bugs) further before the deadline for this project, as we really find it quite interesting to study. But, even without these results we feel that we have learned a great deal about the issues of parallel computing through this project. In particular, it has been interesting to study the implications of latency and bandwidth on a real-world problem. We both feel like we have a much stronger understanding of the algorithmic and mathematical choices that must be made and analyzed to achieve satisfactory results in parallel than we had before doing the work for this project. Finally, having looked at a problem like this from several different angles, and by studying how this sort of problem fits into broader categories of mathematical and parallel work, we have a greater appreciation for the challenges that must be faced when moving from a scientific theory, formula, or model to a full-fledged parallel application.

Acknowledgements

Thank you to Mark Hoemmen for helping us put the issue of *Krylov* subspaces into a meaningful and useful context and to Professor Demmel and everyone else involved with CS267 for inspiration and assistance.

Reference:

1. Prof. Demmel's CS267 Lecture 14 p. 4