Reducing Communication in Distributed Asynchronous Iterative Methods

Jordi Wolfson-Pou and Edmond Chow

School of Computational Science and Engineering,
Georgia Institute of Technology, Atlanta, Georgia, U.S.A.

jwp3@gatech.edu, echow@cc.gatech.edu

Abstract
Communication costs are an important factor in the performance of massively parallel algorithms. We present a new asynchronous parallel algorithm for solving sparse linear systems that reduces communication compared to other algorithms on distributed memory machines. Implemented using passive one-sided remote memory access (RMA) MPI functions, the new method is a variation of the Southwell method, where rows are relaxed greedily, instead of sequentially, by choosing the row with the maximum residual norm. A process relaxes its rows if it holds the maximum residual norm among its neighbors at any given moment. Experimental results show that this method reduces communication costs compared to several other asynchronous iterative methods and the classic synchronous Jacobi method.

Keywords: asynchronous iterative methods, Gauss-Seidel, Southwell, remote memory access

1 Introduction
Inter-node communication has high energy costs compared to computation. In exascale computing, reducing communication, and thus energy costs, may become even more important than reducing overall time. We apply this extreme objective to relaxation-type iterative methods and propose a new asynchronous parallel Southwell method for solving sparse linear systems. This new method uses the idea of asynchronous relaxations, where each process does not have to wait for other processes in order to continue its computation. There have been numerous U.S. Department of Energy reports suggesting that asynchronous computation is necessary to save energy in exascale computations [1–4, 6].

The standard sequential Southwell method relaxes one row at a time, corresponding to the row with the largest residual norm, and is a sequential algorithm by definition. Southwell can converge faster than Gauss-Seidel in terms of the number of relaxations. Since each relaxation is associated with communication, communication cost can also be reduced. However, to achieve this faster convergence, Southwell needs global communication after every relaxation step to determine the row with the largest residual. The goal is to develop a method to solve problems
with less communication than with Gauss-Seidel or Jacobi, with the possibility of also being faster on extreme scale distributed systems.

In our new distributed parallel Southwell algorithm, we approximate the standard Southwell iteration by 1) avoiding global communication, and 2) allowing some rows to be relaxed simultaneously. To describe the method, assume for now one processor node assigned to one row or grid point. At each iteration, each node determines if it is the node with the largest residual among its neighbors in the grid. If so, then this node relaxes its row and sends updated values to its neighbors. We will show that this method reduces communication compared to standard synchronous and asynchronous methods.

Although the theory of asynchronous iterative methods is well-established [8, 9, 11], not much research has been done in distributed implementations that are truly asynchronous, i.e., implementations using operations that allow an origin process to access memory on a target process while the target is occupied with other tasks. The implementation presented in this paper does just this by using passive one-sided asynchronous remote memory access (RMA) operations included in MPI-3 [5].

2 Related Work

Asynchronous iterative methods were first proposed in 1969 by Chazan and Miranker in their paper Chaotic Relaxation [11]. Their paper presented iterative methods that performed relaxations at arbitrary times. These methods included asynchronous Gauss-Seidel and Jacobi methods. Additionally, the paper established a convergence theorem, later expanded upon by Baudet [8]. Baudet also showed experimental results for asynchronous methods implemented on Carnegie Melon’s C.mmp machine [8]. Baudet showed that asynchronous implementations of Jacobi and Gauss-Seidel methods took fewer relaxations and a lower wall clock time to converge, which was also demonstrated by Bull and Freeman [10]. For an overview of asynchronous methods, see [13].

In general, recent research in asynchronous iterative methods have addressed shared memory implementations, while few have conducted studies for reducing communication on distributed systems. An exception is Jager and Bradley [12], who experimentally showed that the asynchronous methods described by Chazan and Miranker had a lower communication cost when implemented with MPI-2 in an asynchronous way. They introduced two new asynchronous algorithms, the Reluctant and the Variable Threshold methods. These two methods used a threshold to determine whether a row should be relaxed based on the change in the row’s solution value after a relaxation or an update from a remote process. Nishida and Kuang [19] suggested a new asynchronous method where updated solution values were sent in bulk after some specified number of relaxations in order to reduce communication overhead. Although implemented on shared memory, Avron, Druinsky and Gupta [7] proposed a method in which rows are randomly relaxed. In general, recent and past work has not fully taken advantage of MPI’s newest asynchronous communication features. Additionally, the amount of research in developing new communication-avoiding asynchronous methods has been sparse.
3 Background

3.1 Jacobi and Gauss-Seidel Methods

The Jacobi and Gauss-Seidel methods are examples of fixed point methods for solving linear systems. A general fixed point iteration method with initial guess \( x^{(0)} \) can be written as

\[
x^{(k+1)} = C x^{(k)} + d,
\]

where the \( C \in \mathbb{C}^{n \times n} \) is the iteration matrix, and \( d, x \in \mathbb{C}^n \). The iteration number \( k = 0, 1, \ldots \) is bounded from above by the stopping criteria for convergence. We define the update of \( x_i^{(k)} \) to \( x_i^{(k+1)} \) as the relaxation of row \( i \).

The Jacobi and Gauss-Seidel methods are fundamental algorithms for solving systems of linear equations \([20]\). For a system \( A x = b \), the Jacobi method can be expressed as

\[
x^{(k+1)} = D^{-1}(b - (L + U)x^{(k)}),
\]

where \( D \) is a diagonal matrix, and \( L \) and \( U \) are the strict lower and upper parts of \( A \), respectively. Therefore, the iteration matrix is \( D^{-1}(L + U) \). Expressed element-wise,

\[
x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right),
\]

where \( i = 1, 2, \ldots, n \).

Instead of using the entire \( x^{(k)} \) vector to update \( x^{(k+1)} \), the Gauss-Seidel method uses the most recent values of \( x \) to update element \( x_i^{(k+1)} \). The method can be expressed as

\[
x^{(k+1)} = (D + L)^{-1}(b - Ux^{(k)}),
\]

where the iteration matrix is \( (D + L)^{-1}U \). Expressed element-wise,

\[
x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j = 1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j = i+1}^{n} a_{ij} x_j^{(k)} \right).
\]

Now let us consider carrying out fixed point iterations in parallel on a distributed memory machine. For simplicity, we assign one row per process. The routine is simple: until convergence, each process \( i \) updates its \( x_i \) value using local data and communicates the update to other processes. We define local as the use of data from shared memory. Therefore, Eq. 5 shows that Gauss-Seidel cannot be implemented in parallel because it uses the most recent information from other processes, which is not in local memory until the next iteration, assuming communication occurs after a phase of computation. Conversely, Eq. 3 is a parallel algorithm by definition because it uses only values from the previous iteration. Extending this to more than one row per process, each process executes a local sweep update before sending data. We define a local sweep update as sequentially relaxing all rows local to a process.

It is important to note that we are dealing with sparse matrices in this paper. The sums in Eqs. 3 and 5, which take on the values \( j = 1, 2, \ldots, n \), should be modified to take advantage of sparsity. Instead of looping over an entire column, the loops should only visit the indices that belong to the neighbor cluster of row \( i \). For row index \( \eta_j \neq i \), we say that row \( \eta_j \) is a neighbor...
Figure 1: Number of row relaxations needed to reach a particular $||r||_2$ for Gauss-Seidel (blue circles), Parallel Southwell (green triangles), and Southwell (red diamonds) using a small problem size, $36 \times 36$ (208 nnz) finite element discretization of the Laplace PDE. We used a random initial guess in the range $[-.5,.5]$ and a 0 right-hand side.

of the current row $i$ if $a_{ij} \neq 0$. We define the neighbor cluster of row $i$ as the set of $q_i$ indices \( \{\eta_1, \eta_2, \ldots, \eta_{q_i}\} \) where each index in the set satisfies the neighbor requirement. In other words, the set of all rows that would be affected by an update from row $i$.

### 3.2 Sequential Southwell Method

Commonly used in computer graphics radiosity problems [15], the Southwell method takes a greedy approach to the order in which rows are relaxed [16,22,23]. Instead of relaxing each row in order, as in Gauss-Seidel, a Southwell relaxation chooses the row $i$ with maximum residual norm and updates the residuals of the remaining rows. A row is relaxed such that $r_i = 0$, i.e., $x_i = x_i + r_i \frac{a_{ii}}{a_{ii}}$. The residuals of the remaining rows can be updated by calculating $r_j = r_j - r_i \frac{a_{ji}}{a_{ii}}$ where $a_i$ is column $i$ of $A$.

The pseudocode for the Southwell iteration method is shown in Algorithm 1. For frame of reference, Gauss-Seidel can be expressed in the same way as shown in Algorithm 2 where $i$ loops from 1 to $n$.

#### Algorithm 1: Southwell Relaxation

1. Determine $i$ s.t. $|r_i|$ is maximized
2. $x_i = x_i + \frac{r_i}{a_{ii}}$
3. For $j = \eta_1, \eta_2, \ldots, \eta_{q_i}$ do
   4. $r_j = r_j - r_i \frac{a_{ji}}{a_{ii}}$
5. End
6. Set $r_i = 0$

**Algorithm 2: Gauss-Seidel Relaxation**

1. $x_i = x_i + \frac{r_i}{a_{ii}}$
2. For $j = \eta_1, \eta_2, \ldots, \eta_{q_i}$ do
   3. $r_j = r_j - r_i \frac{a_{ji}}{a_{ii}}$
4. End
5. Set $r_i = 0$

The advantage of Southwell over Gauss-Seidel is that Southwell often requires a lower number of total rows relaxed in order to reach convergence, as shown in Fig. 1. The test problem used in Fig. 1 is a small finite element Laplace matrix. When translated to a parallel environment, the number of relaxations are correlated with the amount of data communicated between processes. Therefore, an algorithm using the concept of Southwell iterations has the potential
to reduce communication. Unfortunately, in order for a standard Southwell relaxation to take
place, global communication is required to determine the process with the maximum residual
norm.

4 Asynchronous Iterative Methods

4.1 Mathematical Formulation

In general, a simple asynchronous iteration is carried out by 1) reading data from local memory,
2) executing some number of relaxations, and 3) communicating the updates to neighboring
processes. Each process carries out these iterations independent of the progress on other pro-
cesses. Two simplified models of asynchronous iterations are shown in Algs. 3 and 4. Variable
$m_i$ is the number of rows on processor $i$ such that $x_i$ is of size $m_i$ and $\sum_{i=1}^{p} m_i = n$, where $p$ is
the number of processes. Algorithm 3 describes iterations in which updates are communicated
after a single row has relaxed. Algorithm 4 describes iterations in which updates are communicated
after a single sweep through all rows, where package is defined as combining all data into one
message.

**Algorithm 3: Asynchronous Relaxations**

```plaintext
1 while not converged on process $i$ do
2 Read $x_i$ from local memory
3 for $j = 1$ to $m_i$ do
4 Relax row $j$
5 Communicate $x_{i,j}$
6 end
7 end
```

**Algorithm 4: Asynchronous Local Sweeps**

```plaintext
1 while not converged on process $i$ do
2 Read $x_i$ from local memory
3 Relax all $m_i$ rows
4 Package and communicate $x_i$
5 end
```

From Eq. 1, Algs. 3 and 4 can be expressed mathematically as

$$x_i^{(k+1)} = \begin{cases} 
  x_i^{(k)}, & \text{if } i \neq g_i(k), \\
  \sum_{j=0}^{n} c_{ij} x_j^{(f_j(k))} + d_i, & \text{if } i = g_i(k).
\end{cases} \quad (6)$$

The function $g : \mathbb{N} \rightarrow \mathbb{N}$ determines if a row should relax, where $i \in g_i(k)$ for some $k$. Function
$f : \mathbb{N} \rightarrow \mathbb{N}$ maps $k$ to the iteration index of the most recently updated $x$ values, where $f_j(k) \leq k$
and $\lim_{k \to \infty} f_j(k) = \infty$. The second constraint on $f$ indicates that in some number of iterations,
$x_i^{(k+1)}$ will use newer information from other rows. For example, $f_j(k) = k - 1$ and $g_i(k) = k$
mod $(n + 1)$ for Gauss-Seidel iterations.

Chazan and Miranker introduced a general convergence theorem for asynchronous iterative
methods [11], which was later expanded upon by Baudet [8].

**Theorem 1.** Consider the fixed point method from Eq. 1 and the generalized asynchronous
iterative method in Eq. 6. If $\rho(|C|) < 1$, then $x^{(k+1)}$ in Eq. 6 converges to the fixed point $x^*$.

For the purposes of this paper, we can choose the Jacobi iteration matrix for the iteration
matrix $C$, which is the worst case. We can therefore assume guaranteed convergence of our
methods, provided that our test problems satisfy the above theorem.

If there is more than one row per process, a choice must be made for how the local rows will
be relaxed. In this paper, the following asynchronous methods will be compared with our new
method.
• **Asynchronous Gauss-Seidel Local Sweep (AGSLS):** Execute Gauss-Seidel sweeps, package the data, and communicate packaged data to neighbors. This is an example of Alg. 4.

• **Asynchronous Gauss-Seidel Local Relax (AGSLR):** Execute Gauss-Seidel relaxations and communicate updates to neighbors after each relaxation. This is an example of Alg. 3.

• **Asynchronous Southwell Local Relax (ASLR):** Execute sequential Southwell iterations and communicate updates to neighbors after each relaxation. This is an example of Alg. 3, and is a variation of our new method described in Sec. 4.2.

### 4.2 Parallel and Asynchronous Parallel Southwell Methods

We now propose the Parallel Southwell (PS) method. For simplicity, let us assume an equal number of rows and processes, with one row assigned to a process. First, each process checks to see if it holds the maximum residual norm within its neighbor cluster. If so, it relaxes and sends updates packaged with its new residual to its neighbors. Otherwise, it waits for incoming updates while other processes relax their rows. We can simulate this method sequentially, as shown in Figure 1, which compares a simulated PS with Southwell and Gauss-Seidel. The sequential simulation first executes a preliminary sweep in which each row determines whether it should update. An additional sweep is then made that relaxes the rows that calculated themselves as maximum during the first sweep. These rows are relaxed using updates from the previous iteration, as in Jacobi. The figure shows that with respect to number of total relaxations for a given residual norm, PS falls somewhere between Southwell and Gauss-Seidel in terms of total relaxation count for a given residual norm.

We now propose the asynchronous parallel Southwell (APS) method, which is an asynchronous implementation of PS. The APS method is a type of asynchronous local sweep method, and takes on the following steps on processor $i$:

1. Check for incoming data from neighboring processes which consists of solution updates to $\mathbf{x}$ packaged with new residual norms. We define residual norms from neighbors as $r_1, r_2, \ldots, r_q$, and the process’s local residual norm as $r_\ell$. In our algorithm, we used the 2-norm to calculate the residual norms.
2. Determine if $r_\ell = \max(\gamma)$, where $\gamma = \{r_1, r_2, \ldots, r_q, r_\ell\}$.
3. If $r_\ell$ is the max, execute a Gauss-Seidel sweep and communicate solution updates packaged with the new $r_\ell$ to all neighbors. We define outbound solution updates as $\chi_1, \chi_2, \ldots, \chi_q$.

Pseudocode for the routine is shown in Algorithm 5.

**Algorithm 5:** Asynchronous Parallel Southwell method

```plaintext
1 while not converged on process $i$ do
2     Check for updates from neighboring processes
3     Set $\gamma = \{r_1, r_2, \ldots, r_q, r_\ell\}$
4     Calculate $\max(\gamma)$
5     if $r_\ell$ is max then
6         Perform Gauss-Seidel sweep
7         Set $\sigma = \{(\chi_1, r_\ell), (\chi_2, r_\ell), \ldots, (\chi_q, r_\ell)\}$
8         for $j = \eta_1, \eta_2, \ldots, \eta_q$ do
9             Communicate $\sigma(j)$ to neighbor $j$
10     end
11 end
12 end
```

Reducing Communication in Distributed Asynchronous Iterative Methods Wolfson-Pou and Chow
5 Implementation

We used passive one-sided remote memory access (RMA) MPI operations [5] to implement the asynchronous algorithms outlined in section 4. One-sided communication operations can eliminate the requirement for a communication handshake when data is moved between origin and target processes. The two types of one-sided communication are active and passive. For active communication, both processes must be aware of the communication by using blocking or partial blocking commands. For passive communication, only the origin needs to be involved in accessing the data at the target, which is more appropriate for implementing asynchronous algorithms.

An access epoch must be initialized at the target’s memory window in order for the origin process to access data on the target. The origin must also and specify an exclusive or shared lock on that window. If an exclusive lock is specified, only the origin process can access data at the target, while shared allows any number of processes to access the data. There are three primary operations that can be used for data movement: MPI_Put(), MPI_Get(), and MPI_Accumulate(). The function MPI_Put() places data on the target process, MPI_Get() retrieves data, and MPI_Accumulate() updates data on the target using a specified operation. The MPI_Put() and MPI_Get() are problematic for asynchronous iterative methods because of access collisions. Unknown behavior results when multiple origin processes try to alter data on the same target with calls to MPI_Put() or MPI_Get(), so an exclusive lock must be used. This issue is addressed in [14,17,21] by using efficient lock scheduling schemes. For simplicity, this paper bypasses this by using only the MPI_Accumulate() with an MPI_SUM operation, which handles multiple accesses to the same window. Additionally, MPI_Win_lock() with a shared lock is used to make sure that each process can receive data from multiple neighbors at a time.

6 Results

6.1 Test Framework

Our passive one-sided RMA implementations were compiled and run with Intel MVAPICH2 on part of Georgia Institute of Technology’s Partnership for Advanced Computing Environment (PACE) cluster. We used 5 nodes, each with two 10-core Intel Xeon E5-2680 CPUs with InfiniBand interconnect.

The stopping criteria for convergence was determined by locally checking, on each process, if $\|\Delta x_i\|_2 < tol$ or if sweeps > max_sweeps, where $x_i$ is the local solution to a process, $tol$ is some prescribed tolerance, and max_sweeps is an upper bound on the sweeps. If a process satisfies the convergence criteria, it sends a flag to all other processes. On each process, if all flags have been set, then that process terminates.

We choose a random initial guess for $x$ in the range $[-.5,.5]$ and a 0 right-hand side. The test problems we used were finite difference (FD) and finite element (FE) discretizations of the Laplace PDE on a 2D domain. The matrices were stored using compressed sparse column format. The matrices were ordered using the serial graph partitioning software METIS [18]. METIS uses an adjacency matrix of an undirected graph and a desired number of subdomains as input. The goal of METIS is to (1) assign approximately the same number of elements to each subdomain, and (2) partition the graph such that either the edge cut or communication volume between the subdomains is reduced. For parallel computations, the first goal addresses load balancing issues, and the second goal reduces communication between processes. We observed no difference in communication cost when choosing edge cut versus communication.
volume reduction, so we arbitrarily chose edge cut reduction. For this paper, the number of subdomains equals the number of processes.

The metrics studied in this paper are the communication cost and the communication ratio. The communication cost is defined as the total number of messages sent from the first relaxation until the convergence criteria is met. The communication ratio is the ratio of the communication cost of APS to that of some other algorithm.

### 6.2 Comparing All Methods

In this section we compare APS with the asynchronous algorithms defined in Section 4 and synchronous Jacobi (SJ). We used an FE matrix with 349,603 non-zero values and all 100 processes (5 nodes, 20 processes per node) with ∼500 rows per process. We looked at the residual norm, specifically $||r||_2$, as a function of communication cost and wall time.

Figure 2 (a) shows the results for the communication cost, and 2 (b) shows the results for wall time, both in log scale. The results show that the asynchronous sweep methods, i.e., the APS and AGSLS methods, clearly lead in reducing communication and wall time over the other three methods. It is expected that sending after every relaxation would converge faster than Jacobi because updates from other processes should be more readily available at a given point in time. This frequent access to newer data should result in a lower communication cost and wall time because the convergence rate would be faster, i.e., a convergence rate closer to that of Gauss-Seidel. However, it is shown here that this may not be true in practice, seeing as SJ has similar behavior to the two asynchronous relaxation methods, i.e., the ASLR and AGSLR methods. This suggests that the communication overhead associated with sending after each relaxation reduces the potential to increase the convergence rate.

### 6.3 Comparing APS and AGSLS

In this section we look at how APS compares with the next best algorithm, AGSLS. The first test was taking a fixed problem size and varying the number of processes. We used an FE matrix with 1,407,811 non-zero values and varied the number of processes by increments of 20 from 20 to 100, with 20 processes per node. Figure 3 (a) shows the residual norm as a function of the communication cost, for 100 processes. APS clearly reduces communication
Figure 3: Communication cost and ratio comparisons of APS and AGSLS for the FE Laplace matrix test. Plot (a) shows the residual norm as a function of the communication cost for 100 processes. Plot (b) shows the communication ratio for different numbers of processes.

over AGSLS. Figure 3 (b) shows the communication ratio for different numbers of processes. The figure shows that the with a higher number of processes, in this case $p = 80$ and 100, a smaller communication ratio is attainable. For example, for the lowest residual norm, APS with 20 processes costs $\sim 70\%$ of AGSLS’s cost with 20 processes, while APS with 100 processes costs $\sim 50\%$ of AGSLS with 100 processes.

We also varied the problem size and the number of processes, while keeping the number of rows per process constant. We used an FD matrix for this test and fixed 5000 rows per process. The resulting problem sizes ranged from 489,960 to 2,489,800 non-zeroes. Again, Figure 4 (a) shows the residual norm as a function of the communication cost for the largest problem size, i.e., 100 processes with 5000 rows per process. Figure 4 (b) shows the communication ratio for different numbers of processes. This test gives a clearer result, showing that if we maintain the same local problem size, the communication ratio tends to be smaller for a larger number of processes. For example, 80 and 100 processes give the smallest ratio for a sufficiently low residual norm. This result shows that the new method is more advantages for larger numbers of processes. This supports the idea that our method has the potential to greatly reduce communication for exascale computing.

7 Conclusion

In this paper, we proposed an asynchronous parallel Southwell (APS) method. The method is based on the sequential Southwell method, which relaxes the row with largest residual norm on each iteration, instead of in a prescribed order as in Gauss-Seidel. The APS method relaxes the row with the highest residual norm in a neighbor cluster, which allowed for parallelism and eliminated global communication. We showed that this method successfully reduced inter-node communication compared to four other methods. The proposed method may be useful for exascale computing. The ideas in this paper may also apply to computations in sensor networks, where nodes are composed of very simple compute devices with limited energy resources, and communication costs of an in-network computation dominate computational costs, and where overall computation time may not be the highest priority.
Figure 4: Communication cost and ratio comparisons of APS and AGSLS for the FD Laplace matrix test. Plot (a) shows the residual norm as a function of the communication cost for 100 processes. Plot (b) shows the communication ratio for different numbers of processes.

8 Acknowledgments

This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Number DE-SC-0012538. We wish to acknowledge Jeff Hammond for his advice on passive one-sided MPI.

References