# Distributed Southwell: An Iterative Method with Low Communication Costs

Jordi Wolfson-Pou
School of Computational Science and Engineering
College of Computing, Georgia Institute of Technology
Atlanta, Georgia, United States of America
jwp3@gatech.edu

Edmond Chow
School of Computational Science and Engineering
College of Computing, Georgia Institute of Technology
Atlanta, Georgia, United States of America
echow@cc.gatech.edu

## ABSTRACT

We present a new algorithm, the Distributed Southwell method, as a competitor to Block Jacobi for preconditioning and multigrid smoothing. It is based on the Southwell iterative method, which is sequential, where only the equation with the largest residual is relaxed per iteration. The Parallel Southwell method extends this idea by relaxing equation $i$ if it has the largest residual among all the equations coupled to variable $i$. Since communication is required for processes to exchange residuals, this method in distributed memory can be expensive. Distributed Southwell uses a novel scheme to reduce this communication of residuals while avoiding deadlock. Using test problems from the SuiteSparse Matrix Collection, we show that Distributed Southwell requires less communication to reach the same accuracy when compared to Parallel Southwell. Additionally, we show that the convergence of Distributed Southwell does not degrade like that of Block Jacobi when the number of processes is increased.

## KEYWORDS

iterative methods, sparse linear systems, Jacobi, Gauss-Seidel, Southwell, multigrid, reducing communication, one-sided MPI, remote memory access

**ACM Reference format:**
Jordi Wolfson-Pou and Edmond Chow. 2017. Distributed Southwell: An Iterative Method with Low Communication Costs. In *Proceedings of SC17, Denver, CO, USA, November 12–17, 2017,* 13 pages.
DOI: 10.1145/3126908.3126966

## 1 INTRODUCTION

For distributed computing, one of the most commonly used multigrid smoothers is Block Jacobi, where the blocks come from an appropriate partitioning of the problem. This method is highly parallel, but has two main disadvantages: (1) it does not converge for all symmetric positive definite matrices, and (2) convergence degrades when parallelism is increased by using more blocks and thus smaller blocks. On the other hand, Gauss-Seidel converges more rapidly than Block Jacobi and converges for all symmetric positive definite matrices. The disadvantage of Gauss-Seidel is that it is inherently a sequential method. Gauss-Seidel can be parallelized by using block multicoloring, but a large number of colors may be needed for irregular problems [3].

In this paper, our starting point is a related but little-known algorithm called the Southwell method [16, 17]. While Gauss-Seidel can be interpreted as relaxing a set of equations in a specific order, Southwell can be interpreted as relaxing equations one at a time in a dynamic and greedy fashion depending on which equation has the largest residual. In this way, Southwell can converge faster than Gauss-Seidel. However, Southwell is sequential by definition, since the choice made for which equation to relax depends on the previous relaxation. In this paper, we call this method the Sequential Southwell method.

Previously, we introduced the idea of Parallel Southwell [18]. In this method, equation $i$ is relaxed if it has the largest residual compared to the residuals of the equations coupled to variable $i$. This method allows equations to be relaxed simultaneously, and does not require global communication to determine the equation with the largest residual. Ref. [18] explored a simple way of implementing Parallel Southwell in distributed memory using asynchronous communication semantics. In this implementation, communication was reduced by having processes piggy-back its new residual norm (for a block of equations) in messages sent to neighboring processes, which were only sent after a process relaxed its equations. This implementation, however, could deadlock when processes used stale values of the residual norms of their neighbors.

This paper presents the Distributed Southwell method, which addresses the deadlock issue that arises when implementing Parallel Southwell in distributed memory. In Distributed Southwell, each process stores the estimates to its own residual norm that are held by its neighbors. These estimates are used to avoid deadlock. Additionally, each process locally computes better estimates of residual norms that belong to its neighbors without any communication. Importantly, Distributed Southwell also uses new techniques to reduce communication compared to Parallel Southwell in distributed memory. We implement Distributed Southwell using one-sided MPI functions, and run experiments with up to 8192 MPI processes. For our test problems we use a set of test matrices from the SuiteSparse matrix collection[7]. We show that Distributed Southwell usually requires far less communication to converge when compared with Parallel Southwell. Additionally, we show that the convergence of Distributed Southwell does not degrade like that of Block Jacobi when the number of processes is increased.

## 2 BACKGROUND

### 2.1 Stationary Iterative Methods

The Jacobi and Gauss-Seidel methods are fundamental stationary iterative methods for solving the sparse, linear system $Ax = b$[15]. A general stationary iterative method, with initial guess $x^{(0)}$, can be written as

$$x^{(k+1)} = Gx^{(k)} + f, \tag{1}$$

where $G$ is the iteration matrix, $f$ is a vector, and the bracketed superscript is the iteration index. The residual vector corresponding to the approximation $x^{(k)}$ is $r^{(k)} = b - Ax^{(k)}$.

For a given stationary iterative method, we define the update of the $i^{\text{th}}$ component from $x_i^{(k)}$ to $x_i^{(k+1)}$ as the *relaxation* of row $i$. For a system with $n$ equations, we define the relaxation of $n$ rows as a *sweep*. We further define a *parallel step* as a phase of computation in which rows are relaxed in parallel. For example, a sweep of Jacobi is also a parallel step of Jacobi. For sequential Gauss-Seidel, each parallel step relaxes a single equation, while for Multicolor Gauss-Seidel, a parallel step involves relaxing all rows belonging to a single color. For example, if a problem has seven colors, it takes seven parallel steps of Multicolor Gauss-Seidel to perform one sweep.

### 2.2 The Sequential Southwell Method

We first introduce some notation necessary for explaining the Sequential Southwell method and its parallel variants. For row $i$, row indices $\eta_j \neq i$ are *neighbors* of $i$ if $a_{\eta_j i} \neq 0$. We define the *neighborhood* of row $i$ as the set $N_i = \{\eta_1, \eta_2, \ldots, \eta_{q_i}\}$ of cardinality $q_i$, where each index in the set is a neighbor, i.e., the neighborhood of row $i$ is the set of rows coupled with row $i$. We also define $\Gamma_i = \{|r_{\eta_1}|, |r_{\eta_2}|, \ldots, |r_{\eta_{q_i}}|\}$ where $r_{\eta_j}$ is the residual of equation $\eta_j$.

Instead of relaxing rows in some prescribed order as in the Gauss-Seidel method, each step of Sequential Southwell relaxes the row $i$ with the largest component of the residual vector. Then the residual vector is updated, but notice that only components corresponding to neighbors of row $i$ need to be updated. Formally,

$$x_i^{(k+1)} = \begin{cases} x_i^{(k)} + \dfrac{r_i^{(k)}}{a_{ii}}, & \text{if } |r_i| \text{ is the maximum for all } i, \\ x_i^{(k)}, & \text{otherwise,} \end{cases} \tag{2}$$

$$r_{\eta_j}^{(k+1)} = r_{\eta_j}^{(k)} - r_i^{(k)} \frac{a_{\eta_j i}}{a_{ii}}, \text{ for all } \eta_j \in N_i,\ j = 1, \ldots, q_i. \tag{3}$$

We also have that the updated residual $r_i^{(k+1)} = 0$ for the row $i$ that was chosen to be relaxed. Note that in this paper, we are technically using the Gauss-Southwell method, which is more natural to analyze, and which was actually first proposed by Gauss. In this method, we relax the row $i$ with the largest $|r_i/a_{ii}|$. The method is identical to what we are calling the Sequential Southwell method when we scale the systems, as we do in this paper, such that the matrices have unit diagonals.

Sequential Southwell can converge faster than Gauss-Seidel in terms of the number of relaxations. However, the method never caught on for automatic computers due to the relatively high cost of determining the row with the largest residual, compared to simply cycling through all equations as in the Gauss-Seidel method. Nevertheless, it has recently found application as an *adaptive* multigrid smoother [14, 13], as a *greedy* multiplicative Schwarz method (where the subdomain with the largest residual norm is chosen to be solved next) [10], as a way of accelerating coordinate descent optimization methods for big data problems [12], and as a scheme for choosing basis vectors when finding sparse solutions to underdetermined inverse problems, e.g., [5, 9].

Our motivation to study and develop Southwell-like methods is due to today's high cost of interprocessor communication compared to computation. Assume for the moment that each parallel process is responsible for a single row of the matrix equation $Ax = b$. When a row is relaxed, that process must send data to the processes corresponding to neighboring rows in order for these processes to update their residuals (see formula (3)). Therefore, each relaxation is associated with communication. If Southwell-like methods reduce the number of relaxations required to solve a problem compared to that of stationary iterative methods, then they also can reduce the amount of communication.

### 2.3 The Parallel Southwell Method

In the Parallel Southwell method[18], instead of relaxing a single row at each step, multiple rows can be relaxed simultaneously, corresponding to rows that have the largest residual within its neighborhood, i.e., row $i$ is relaxed if $|r_i|$ is maximal in $\{\Gamma_i, |r_i|\}$. We define this condition as the *Parallel Southwell criterion.* A geometric illustration of one parallel step is shown in Figure 1.

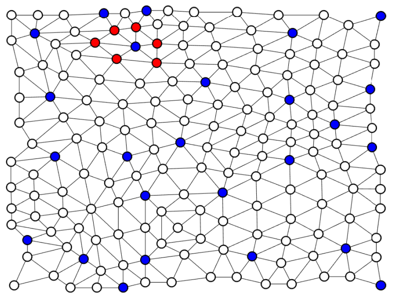


**Figure 1: Illustration of one parallel step of Parallel Southwell. The blue points correspond to the rows chosen to be relaxed in one parallel step via the Parallel Southwell criterion. The red points are neighbors of one of the blue points.**

Figure 2 shows an example of the convergence behavior of Parallel Southwell compared to that of other methods. The matrix is from a finite element discretization of the Poisson equation on a square domain. Irregularly structured linear triangular elements are used. The discrete right-hand side has elements sampled from a uniform random distribution with mean zero and is scaled such that its 2-norm is 1. The example problem has 3081 rows and the convergence for three sweeps of each method is shown.

The figure shows that Sequential Southwell converges fastest, in terms of number of relaxations, compared to other methods. In particular, it requires about half the number of relaxations as Gauss-Seidel when only low accuracy is required (e.g., residual norm reduction to 0.6). Parallel Southwell converges almost as rapidly as Sequential Southwell but is of course a parallel method. The markers along each curve delineate the parallel steps. Multicolor Gauss-Seidel requires 6 colors (the number of rows assigned each color is very unbalanced although we assign colors using a breadth-first traversal). For low accuracy, Parallel Southwell requires a much smaller number of relaxations than Multicolor Gauss-Seidel and approximately the same number of parallel steps. Jacobi converges slowest compared to the other methods when convergence is measured in terms of number of relaxations.

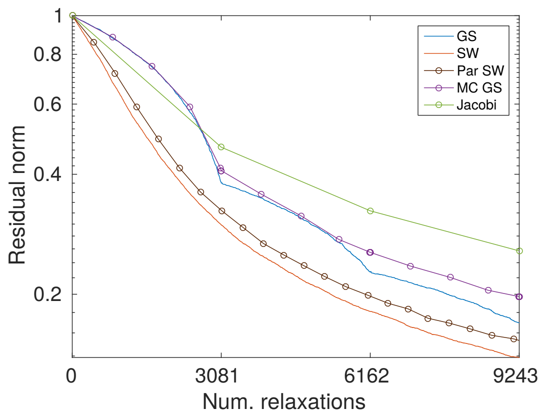


**Figure 2: Convergence for a small finite element problem. The methods compared are Gauss-Seidel (GS), Sequential Southwell (SW), Parallel Southwell (Par SW), Multicolor Gauss-Seidel (MC GS), and Jacobi. The markers along the curves for the parallel methods delineate the parallel steps.**

## 2.4 Block Methods on Distributed Memory Computers

For the Jacobi and Parallel Southwell methods on a distributed memory machine, it is natural to partition a problem into non-overlapping subdomains, with one subdomain for each process. To approximately solve the local subdomain problems, Gauss-Seidel may be used. In the case of Jacobi, this is often referred to as Hybrid Gauss-Seidel[4, 6], or Processor Block Gauss-Seidel[2].

We use the following notation:

- Each parallel process with rank $p$ (ranging from 0 to $\mathcal{P}-1$, where $\mathcal{P}$ is the total number of processes) is responsible for $m_p$ rows of the $n$ total rows, where the partitioning is determined, e.g., with METIS [11].
- The values $\{\delta_0, \delta_1, \ldots, \delta_{\mathcal{P}}\}$ are the $\mathcal{P}+1$ row index offsets, i.e., the prefix sum of $\{0, m_0, m_1, \ldots, m_{\mathcal{P}-1}\}$.
- Each process stores $r_p$ and $x_p$ corresponding to the $\vec{\delta}_p = \delta_p : (\delta_{p+1}-1)$ (Matlab array notation) portion of the global residual and solution arrays, respectively.
- We use the one-sided memory model, where $p$ has a region of memory that remote processes can directly write to without the involvement of $p$. We define this region of memory as the *memory window* $\mathcal{W}_p$ of $p$.

In this notation, the Block Jacobi algorithm is shown in Algorithm 1.

**Algorithm 1:** Block Jacobi

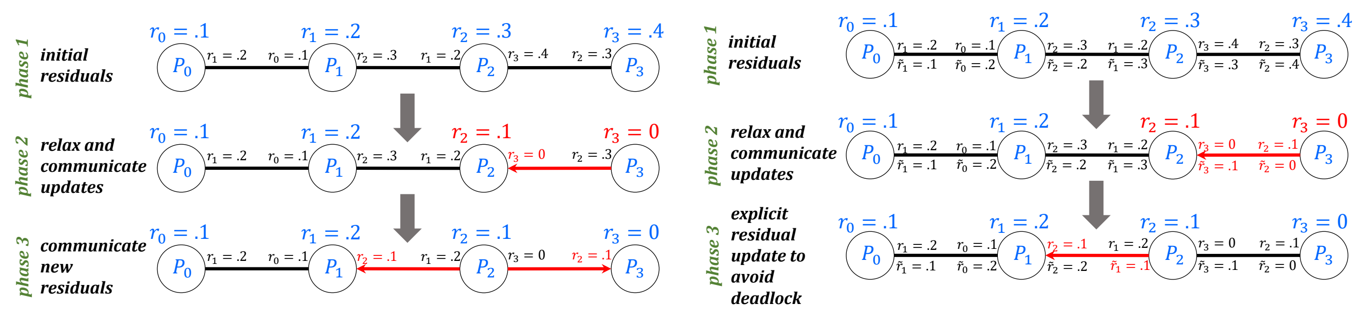
```
1  Set $r = b - Ax$
2  for each process with rank $p$ do
3      Set $r_p = r(\vec{\delta}_p)$
4      Set $x_p = x(\vec{\delta}_p)$
5  end
6  for $k = 1, \ldots, k_{max}$ on process with rank $p$ do
7      Update $x_p$ and $r_p$ by relaxing the equations belonging to $p$
8      Write updates to $\{\mathcal{W}_1, \ldots, \mathcal{W}_{q_p}\}$
9      Wait for neighbors to finish writing to $\mathcal{W}_p$
10     Read from $\mathcal{W}_p$ to update $r_p$
11 end
```

For the block form of Parallel Southwell, instead of comparing the magnitude of individual residual vector components, we now compare the residual norm for the rows of process $p$ to the residual norms for the rows that belong to the neighbors of $p$, i.e., we redefine $\Gamma_p = \{\|r_1\|_2, \|r_2\|_2, \ldots, \|r_{q_p}\|_2\}$ where we assume the neighboring processes have indices $1, 2, \ldots, q_p$. If process $p$ satisfies the Parallel Southwell criterion, it relaxes the equations in its subdomain and sends updates to its neighbors. Upon receiving these updates, the neighbors of $p$ use this new information to update their boundary points. Additionally, at each parallel step, an extra communication step is needed in order for $p$ to know the residual norms that belong to its neighbors. We call this an *explicit residual update*, where a process sends its updated residual norm to its neighbors.

A geometric interpretation of the block version of Parallel Southwell is shown in Figure 3, where the top mesh shows the partitioning, and the bottom mesh shows the subdomains that are selected to be updated via the Parallel Southwell criterion.

An illustration of the key phases of a parallel step of Parallel Southwell is shown in Figure 4(a). The illustration shows four processes, where the edges connecting them indicate a neighbor relationship. In phase 1 of the figure, $p_3$ is the only process that determines that it must update. In phase 2, $p_3$ updates, and writes to the memory of $p_2$, which counts as a single message. This changes the residual of $p_2$, and updates the copy of the residual of $p_3$ held by $p_2$. In phase 3, $p_2$ detects that its own residual has changed, so it updates the copies of its residual that $p_1$ and $p_3$ hold, which requires two additional messages to be sent.

Algorithm 2 shows the block form of Parallel Southwell implemented in distributed memory. To be clear, this algorithm is mathematically identical to Parallel Southwell implemented in shared memory. Note that this algorithm is different than that introduced in Ref. [18], which can possibly deadlock. To observe how that algorithm might deadlock, consider again Figure 4(a) but now assume that $p_2$ does not communicate its updates, i.e., if we remove the explicit residual norm update from the last phase of the diagram. Deadlock will now occur. This can be seen at phase 2, where the true residual norm of each process ($r_i$ in blue shown above each node) is less than its copies of the residual norm of its neighbors ($r_i$ in black on the right and left of each node, above the connection edge), resulting in all processes failing to satisfy the Parallel Southwell criterion.


phase 1
initial residuals
$r_0 = .1$ $r_1 = .2$ $r_2 = .3$ $r_3 = .4$
$P_0$ $P_1$ $P_2$ $P_3$
phase 2
relax and communicate updates
$r_2 = .1$ $r_3 = 0$
phase 3
communicate new residuals
phase 3
explicit residual update to avoid deadlock

(a) Parallel Southwell parallel step

(b) Distributed Southwell parallel step

**Figure 4: Illustration of a parallel step of (a) Parallel Southwell and (b) Distributed Southwell. In the illustration, a line of four processes $P_0, \ldots, P_3$, with an array communication topology, start the parallel step with exact residuals $r_0, \ldots, r_3$ (shown in blue above the corresponding $P$), and their estimates of the residual norms of their neighbors (shown in black above the inter-process connections). Additionally, in (b), each $P_i$ also stores the estimate of the residual norm of $P_i$ stored by the neighbors of $P_i$, denoted by $\tilde{r}_0, \ldots, \tilde{r}_3$. Each line of four processes, three lines in total, denotes a phase of the parallel step. Red residuals denote an updated residual, and red arrow connections denote communication. Note that the illustration is not based on any data taken from any real experiments.**

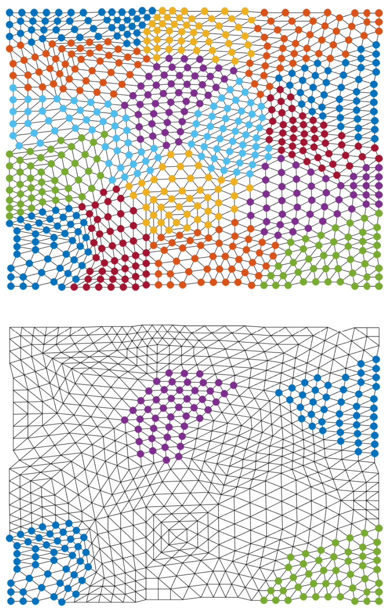


**Figure 3: Parallel Southwell with multiple equations per process. Top: the subdomains assigned to each process. Bottom: four subdomains selected via the Parallel Southwell criterion.**

There are a few communication-reducing optimizations shown in Algorithm 2. First, if process $p$ does not relax its rows, and none of its neighbors relax their rows, there is no need for $p$ to send its residual to its neighbors because its residual has not changed. This is shown in the `If` statement on line 19. Second, if $p$ does satisfy the Parallel Southwell criterion, it can append its new residual norm to all outgoing messages, which eliminates the need to communicate its new residual to its neighbors in a separate message. This is shown in line 10.

**Algorithm 2:** Parallel Southwell (block version)

```
1  Set $r = b - Ax$
2  for each process with rank $p$ do
3      Set $r_p = r(\vec{\delta}_p)$
4      Set $x_p = x(\vec{\delta}_p)$
5      Set $\Gamma_p = \{\|r_1\|_2, \ldots, \|r_{q_p}\|_2\}$
6  end
7  for $k = 1, \ldots, k_{max}$ on process with rank $p$ do
8      if $\|r_p\|_2$ is maximum in $\{\Gamma, \|r_p\|_2\}$ then
9          Update $x_p$ and $r_p$ by relaxing the equations belonging to $p$
10         Write updates and $\|r_p\|_2$ to $\{\mathcal{W}_1, \ldots, \mathcal{W}_{q_p}\}$
11     else
12         Wait for neighbors to finish writing to $\mathcal{W}_p$
13         for $j = 1, \ldots, q_p$ do
14             if Neighbor $q_j$ has written new information to $\mathcal{W}_p$ then
15                 Read from $\mathcal{W}_p$ to update $r_p$
16                 Update $\|r_{q_j}\|_2$ in $\Gamma$
17             end
18         end
19         if $\|r_p\|_2$ has changed then
20             Write $\|r_p\|_2$ to $\{\mathcal{W}_1, \ldots, \mathcal{W}_{q_p}\}$
21         end
22     end
23     Wait for neighbors to finish writing to $\mathcal{W}_p$
24     for $j = 1, \ldots, q_p$ do
25         if Neighbor $q_j$ has written new information to $\mathcal{W}_p$ then
26             Update $\|r_{q_j}\|_2$ in $\Gamma$
27         end
28     end
29 end
```

## 3 THE DISTRIBUTED SOUTHWELL METHOD

The premise of the Distributed Southwell method is that the residuals for the equations on neighboring processes do not need to be known exactly. These residuals are only needed for processes to determine if they should relax their own equations. That this step is done precisely following the Parallel Southwell criterion is not essential.

This premise allows many possibilities for reducing communication. In particular, processes do not need to carry out explicit residual updates every time their residual norm changes. Instead, a process $p$ can maintain estimates of the residuals for the equations on neighboring processes. When a process $p$ relaxes its own equations, it knows how these relaxations affect the residual on its neighbor $q$, without any communication. Referring to formula (3), the update

$$-r_i^{(k)} \frac{a_{\eta_j i}}{a_{ii}}$$

to the neighbor residual $r_{\eta_j}^{(k)}$ only depends on local information, in particular $r_i^{(k)}$, while $a_{\eta_j i}$ and $a_{ii}$ are matrix data that can be stored locally (i.e., the process responsible for row $i$ stores column $i$ of $A$).

Consider now a neighbor $s$ of $q$, so that the matrix dependencies are $p \Longleftrightarrow q \Longleftrightarrow s$. If a neighbor $s$ of process $q$ relaxes its equations, then the effect on the residual of $q$ will not be known to process $p$. This is how the estimate that process $p$ has of the residual norm of process $q$ loses accuracy. Again referring to formula (3), the size of the discrepancy is related to the size of the residual component, i.e., it decreases in size as the iterations progress.

As explained in Section 2.4, there is a major drawback of using inaccurate residuals. If the residual norm estimates on all processes is such that no process thinks it has the largest residual norm, then deadlock occurs. This means that there is a risk of deadlock if the residual norm estimates are larger than the actual residual norms. Fortunately, this situation can be detected by a process $q$ maintaining a copy of the estimate that $p$ has of the residual norm of $q$. This copy can be maintained, like above, without communication. Thus $q$ can detect if the estimates of its residual norm are larger than its actual residual norm. In this case, $q$ sends $p$ an explicit message to update its estimate. Deadlock is thus avoided.

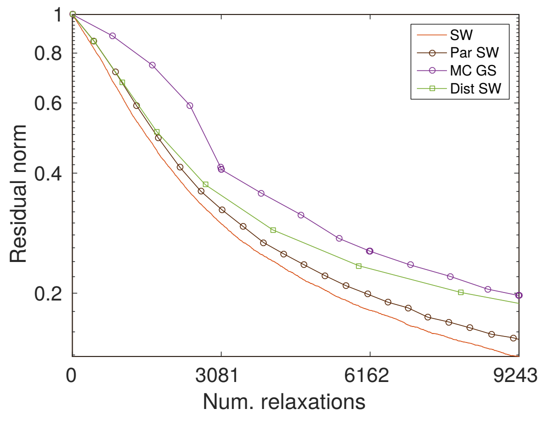


**Figure 5: Convergence for a small finite element problem. Distributed Southwell is compared to other methods (all in scalar form). The markers along the curves for the parallel methods delineate the parallel steps.**

Figure 5 shows the convergence of Distributed Southwell for the same finite element problem as used in Figure 2. The convergence curves for Sequential Southwell, Parallel Southwell, and Multicolor Gauss-Seidel are repeated from that figure for comparison. All methods in this figure use their scalar forms (i.e., subdomain size of 1). We observe that the behavior of Distributed Southwell closely matches that of Parallel Southwell (which uses the exact Parallel Southwell criterion for choosing which equations to relax) for low levels of accuracy (e.g., residual norm 0.6), which is the "sweet spot" for using Southwell-like methods compared to using Gauss-Seidel. We also observe that with inexact residual estimates, Distributed Southwell relaxes more equations per parallel step, as shown by the markers along the curves in the figure. This may account for the degraded convergence of Distributed Southwell compared to Parallel Southwell as more parallel steps are taken.

The Distributed Southwell idea can be easily extended to use subdomains in a practical distributed code. Here, process $p$ stores a ghost layer of residuals corresponding to all off-processor connections to the boundary points of $p$, where $z_{q_j}$ denotes the residual ghost layer for the points $\beta_{q_j}$ of neighbor $q_j$. When process $p$ relaxes its equations, it also updates all points in the ghost layer, and uses this to update all the residual norms in $\Gamma$. These updates denote the contribution of $p$ to the residual norm of its neighbors. This allows $p$ to store more accurate copies of the residual norms of its neighbors, in the case that $p$ updates often without receiving updates from neighbors. When $p$ receives updates from neighbors, the values in the ghost layer and $\Gamma$ are corrected.

In addition to $p$ storing $\Gamma$, $p$ also stores $\tilde{\Gamma}$, which are the residual norms of $p$ stored by the neighbors of $p$. When neighbor $q_1$ of $p$ updates and writes to the memory of $p$, it includes its new estimate of the residual of $p$ in the message. In the memory of $p$, this is the value $\|\tilde{r}_{q_1}\|_2$. This value is always exactly known by $p$, since only $p$ and $q_1$ alter the estimate of the residual norm of $p$ stored by $q_1$. If $p$ determines that $\|\tilde{r}_{q_1}\|_2 > \|r_p\|_2$, then there is a possibility of deadlock, and $p$ communicates its residual norm and boundary points to $q_1$, which brings the estimate of the residual norm of $p$ stored by $q_1$ up to date.

The algorithm for Distributed Southwell (in block or subdomain form) is shown in Algorithm 3. An illustration of the key phases of a parallel step of Distributed Southwell is shown in Figure 4(b). As in (a), $p_3$ updates, but also updates its estimate of the residual norm of $p_2$, obtaining the new residual norm of $p_2$ exactly. If $p_1$ were to also update the residual norm of $p_2$ in this phase, $p_3$ would not have an exact estimate of the residual norm of $p_2$, but it would be a better estimate than if $p_3$ did nothing at all. In phase 3, $p_2$ detects possible deadlock on $p_1$, and sends a single message that updates the estimate of the residual norm of $p_2$ stored by $p_1$.

Our distributed implementations (for all algorithms including Distributed Southwell) use the one-sided semantics provided in MPI-3, also known as remote memory access (RMA)[1]. For one-sided, an origin process writes directly to the memory of a target process without the target being involved in the transfer of data, avoiding the communication required by the receiving side. Another reason we use one-sided functions is that in Parallel and Distributed Southwell, it is not always clear when, or if, a process should expect a message from a neighbor. Each process initially makes an MPI group

**Algorithm 3:** Distributed Southwell (block version)

```
Set $r = b - Ax$
for each process with rank $p$ do
    Set $r_p = r(\vec{\delta}_p)$
    Set $x_p = x(\vec{\delta}_p)$
    Set $\Gamma_p = \{\|r_1\|_2, \ldots, \|r_{q_p}\|_2\}$
    Set $\tilde{\Gamma}_p = \{\|\tilde{r}_1\|_2, \ldots, \|\tilde{r}_{q_p}\|_2\}$
    for $j = 1, \ldots, q_p$ do
        Set $z_{q_j} = r(\beta_{q_j})$
    end
end
for $k = 1, \ldots, k_{max}$ on process with rank $p$ do
    if $\|r_p\|_2$ is maximum in $\{\Gamma_p, \|r_p\|_2\}$ then
        Update $x_p$ and $r_p$ by relaxing the equations belonging to $p$
        for $j = 1, \ldots, q_p$ do
            Update $z_{q_j}$ and compute $\|r_{q_j}\|_2$
            Set $\|\tilde{r}_{q_j}\|_2 = \|r_p\|_2$
            Write updates, $z_p$, $\|r_p\|_2$, and $\|r_{q_j}\|_2$ to $\mathcal{W}_{q_j}$
        end
    end
    Wait for neighbors to finish writing to $\mathcal{W}_p$
    for $j = 1, \ldots, q_p$ do
        if Neighbor $q_j$ has written new information to $\mathcal{W}_p$ then
            Update $r_p$
            Overwrite $z_{q_j}$
            Overwrite $\|r_{q_j}\|_2$ in $\Gamma$ and $\|\tilde{r}_{q_j}\|_2$ in $\tilde{\Gamma}$
        end
        if $\|r_p\|_2 < \|\tilde{r}_{q_j}\|_2$ then
            Set $\|\tilde{r}_{q_j}\|_2 = \|r_p\|_2$
            Write $z_{q_j}$, $\|r_p\|_2$ and $\|r_{q_j}\|_2$ to $\mathcal{W}_{q_j}$
        end
    end
    Wait for neighbors to finish writing to $\mathcal{W}_p$
    for $j = 1, \ldots, q_p$ do
        if Neighbor $q_j$ has written new information to $\mathcal{W}_p$ then
            Overwrite $z_{q_j}$
            Overwrite $\|r_{q_j}\|_2$ in $\Gamma$ and $\|\tilde{r}_{q_j}\|_2$ in $\tilde{\Gamma}$
        end
    end
end
```

consisting of its neighbors, and calls `MPI_Win_allocate()` to create a memory window, i.e., allocates a region of memory that is accessible by remote processes. During communication phases, assuming all processes have information to send, processes enter access epochs by calling `MPI_Win_post()` followed by `MPI_Win_start()`. Messages are then sent using `MPI_Put()`, and the epochs are ended using `MPI_Win_complete()` followed by `MPI_Win_wait()`. A process and all its neighbors must collectively call all commands for starting and ending the access epochs.

## 4 RESULTS

### 4.1 Multigrid Smoothing

We first test the use of Distributed Southwell as a smoother for the multigrid method. Here, we use a scalar rather than block version of Distributed Southwell. The test problem is the 2D Poisson equation on a square discretized on a regular mesh by centered finite differences. The discrete right-hand side is chosen to be a vector with random entries uniformly distributed between -1 and 1. To test multigrid convergence, the grid dimensions are increased from $15 \times 15$ to $255 \times 255$. Each V-cycle uses multiple levels such that the coarsest level corresponds to a $3 \times 3$ grid, at which an exact solve is used.

Each V-cycle uses one step of pre-smoothing and one step of post-smoothing. As a baseline for comparison, we use Gauss-Seidel as a smoother. For Distributed Southwell as a smoother, we use a number of relaxations corresponding to exactly the number of relaxations as Gauss-Seidel (i.e., the number of unknowns in the grid at a given level, called "1 sweep"). We also test Distributed Southwell using half of the number of relaxations of Gauss-Seidel (called "1/2 sweep"). Distributed Southwell selects many rows to be relaxed simultaneously in a single parallel step. In order to achieve an exact total number of relaxations (for our comparison purposes), in the final parallel step of Distributed Southwell, a random subset of the rows selected to be relaxed are actually relaxed.

Figure 6 shows the residual norm relative to the initial residual norm after 9 V-cycles. The most important result is that Distributed Southwell as a smoother shows grid-size independent convergence, even though some rows may never have been relaxed in the smoother, which is particularly true in the case of "1/2 sweep." We also observe that Distributed Southwell is a more efficient smoother than Gauss-Seidel, resulting in better multigrid convergence even when Distributed Southwell uses the same number of relaxations as Gauss-Seidel.

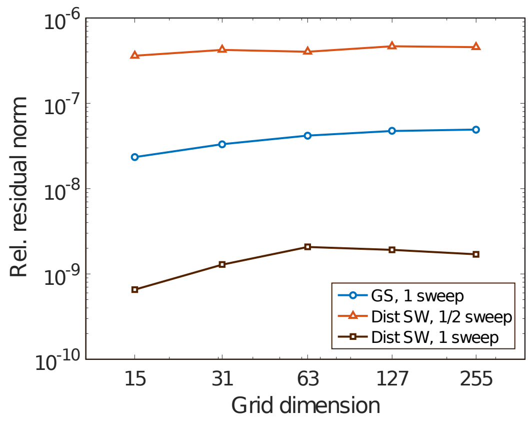


**Figure 6: Relative residual norm after 9 V-cycles of multigrid applied to solving the 2D Poisson equation for increasing grid dimensions. Distributed Southwell as a smoother is compared to Gauss-Seidel (GS) as a smoother. The results show that convergence is independent of grid size in all cases. In addition, Distributed Southwell is more efficient as a smoother, per relaxation, than Gauss-Seidel.**

### 4.2 Test Framework

In the following experiments, we compare Distributed Southwell, Parallel Southwell, and Block Jacobi implemented in distributed memory. Here, we used a random initial guess and a right-hand side $b = 0$. We scaled all initial guesses such that $\|r^{(0)}\|_2 = 1$. All test matrices are shown in Table 1, which were taken from the SuiteSparse Matrix Collection[7], and symmetrically scaled to have unit diagonal values. We used up to 256 32-core nodes on the NERSC Cori (Phase I) supercomputer. We varied the number of parallel steps from zero to 50, and took 50 samples at each parallel step. Out of 50 samples, we used the run that gave us the lowest wall-clock time, i.e., we considered the best time a method could obtain at a given parallel step.

**Table 1: Test problems from the SuiteSparse Matrix Collection. All matrices are symmetric positive definite.**

| Matrix | Number of Non-zeros | Number of Equations |
|---|---:|---:|
| Flan_1565 | 114,165,372 | 1,564,794 |
| audikw_1 | 77,651,847 | 943,695 |
| Serena | 64,122,743 | 1,382,121 |
| Geo_1438 | 60,169,842 | 1,371,480 |
| Hook_1498 | 59,344,451 | 1,468,023 |
| bone010 | 47,851,783 | 986,703 |
| ldoor | 42,451,151 | 909,537 |
| boneS10 | 40,878,708 | 914,898 |
| Emilia_923 | 40,359,114 | 908,712 |
| inline_1 | 36,816,170 | 503,712 |
| Fault_639 | 27,224,065 | 616,923 |
| StocF-1465 | 20,976,285 | 1,436,033 |
| msdoor | 19,162,085 | 404,785 |
| af_5_k101 | 17,550,675 | 503,625 |

For all methods, when a process updates, a single Gauss-Seidel sweep is carried out on the subdomain that the process is responsible for. We note that a single process per node could be used, with a multi-threaded local solver, e.g., Multicolor Gauss-Seidel. Another important note is that we are using the Parallel Southwell method as defined in Section 2.3, and not as defined in [18]. This is because Parallel Southwell as defined in [18] deadlocks for all our test problems.

## 4.3 Reducing $\|r\|_2$ to 0.1 Using 8192 Processes

Table 2 shows results for reducing $\|r\|_2$ to 0.1 with 8192 MPI processes (256 nodes). The table shows the wall-clock time, communication cost, number of parallel steps, number of relaxations, and the number of active processes. "Communication cost" is defined as the total number of messages sent by all processes, divided by the total number of processes. "Active processes" is defined as the average fraction of processes carrying out block relaxations of local subdomains at each parallel step.

The table shows that Block Jacobi can achieve $\|r\|_2 = 0.1$ for only three of the test matrices. Figure 7 plots the convergence with respect to different axes for four problems. In the case of bone010, Block Jacobi initially reduces the residual norm, but eventually diverges. This can also be seen for Geo_1438 and Hook_1498, which are two cases where Block Jacobi can reach the target residual norm. This divergence underscores the unreliability of Block Jacobi, especially when a large number of processes is used. Matrix af_5_k101 is the only case in which Block Jacobi never diverged.

Table 2 also shows the superiority of Distributed Southwell over Parallel Southwell. Distributed Southwell is approximately twice as fast, requires close to a third of the communication, and converges in fewer parallel steps. Parallel Southwell requires fewer relaxations, but needs to communicate more per relaxation.

The fact that Parallel Southwell requires fewer relaxations but requires almost three times the communication shows how costly the explicit residual updates of Parallel Southwell are. This cost is shown in Table 3, where the explicit residual updates by Parallel Southwell dominate the overall communication cost.

We also observe that in Distributed Southwell, more processes are active compared to Parallel Southwell. This is a result of using inexact residual norms. We note that if adjacent subdomains relax at the same time (rather than an independent set of subdomains), then convergence is at risk.

Since multigrid smoothing and preconditioning only requires a small number of sweeps, it is useful to look at the costs per parallel step. This is shown in Table 4, where Distributed Southwell is faster than the other methods.

**Table 3: Communication cost breakdown for Parallel Southwell (PS) and Distributed Southwell (DS), where "Solve comm" denotes the communication cost of sending updates to neighbors after a local subdomain is solved, and "Res comm" denotes the communication cost of explicit residual updates.**

| | Solve comm | | Res comm | |
|---|---:|---:|---:|---:|
| Matrix | PS | DS | PS | DS |
| Flan_1565 | **27.945** | 28.961 | 308.240 | **85.836** |
| audikw_1 | **28.630** | 30.634 | 454.749 | **126.932** |
| Serena | **27.399** | 27.748 | 367.147 | **97.391** |
| Geo_1438 | **26.485** | 27.076 | 325.846 | **86.910** |
| Hook_1498 | **24.076** | 25.744 | 284.861 | **78.220** |
| bone010 | **27.965** | 28.617 | 355.249 | **95.326** |
| ldoor | **11.980** | 12.745 | 69.063 | **20.043** |
| boneS10 | **18.462** | 19.433 | 157.410 | **45.862** |
| Emilia_923 | † | **32.101** | † | **102.375** |
| inline_1 | **24.352** | 27.311 | 298.601 | **77.505** |
| Fault_639 | † | **27.785** | † | **98.213** |
| StocF-1465 | **24.188** | 25.208 | 308.056 | **85.529** |
| msdoor | **11.560** | 12.669 | 76.695 | **23.670** |
| af_5_k101 | **10.603** | 10.947 | 58.091 | **16.651** |

## 4.4 Strong Scaling

We first look at a target residual norm of $\|r\|_2 = 0.1$ and varying the number of MPI processes. Six examples are shown in Figure 8, where wall-clock time is shown as a function of the number of MPI processes. In most cases, and for all methods, the wall-clock time initially decreases as we increase the number of MPI processes, and then starts to increase. This is due to the local subdomain solves, where a single Gauss-Seidel sweep is used. This operation has the complexity of a sparse matrix-vector product, and as the number of MPI processes increases (i.e., local subdomain sizes decrease), the time spent on communication increasingly outweighs the time spent on computation. It can be observed that the poor scalability is worse for the smaller problems.

We can see that Distributed Southwell is always faster than Parallel Southwell, except for Flan_1565 on 64 processes, where

**Table 2: Comparison of Distributed Southwell (DS) with Parallel Southwell (PS) and Block Jacobi (BJ) for reducing the residual to $\|r\|_2 = 0.1$. Linear interpolation on $\log_{10}(\|r\|_2)$ was used to extract this data. The † symbol indicates that a method could not achieve $\|r\|_2 \le 0.1$ in 50 parallel steps. The wall-clock time was determined by taking the minimum of 50 samples, i.e., showing each method performing at its best. "Communication cost" is defined as the total number of messages sent by all processes divided by the number of processes. "Active processes" is defined as the average fraction of processes carrying out block relaxations of local subdomains at each parallel step.**

| | Wall-clock time | | | Communication cost | | | Parallel steps | | | Relaxations/n | | | Active processes | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Matrix | BJ | PS | DS | BJ | PS | DS | BJ | PS | DS | BJ | PS | DS | BJ | PS | DS |
| Flan_1565 | † | 0.547 | **0.234** | † | 336.185 | **114.797** | † | 46.073 | **35.000** | † | **2.249** | 2.330 | † | **0.049** | 0.066 |
| audikw_1 | † | 1.100 | **0.434** | † | 483.379 | **157.566** | † | 44.907 | **33.699** | † | **1.613** | 1.737 | † | **0.036** | 0.049 |
| Serena | † | 0.731 | **0.301** | † | 394.546 | **125.139** | † | 44.193 | **31.764** | † | **1.818** | 1.839 | † | **0.041** | 0.057 |
| Geo_1438 | **0.068** | 0.577 | 0.224 | **53.835** | 352.331 | 113.986 | **3.805** | 44.381 | 30.896 | 3.805 | **1.872** | 1.916 | 1.000 | **0.042** | 0.061 |
| Hook_1498 | **0.064** | 0.523 | 0.234 | **41.335** | 308.938 | 103.964 | **3.040** | 37.368 | 29.495 | 3.040 | **1.809** | 1.939 | 1.000 | **0.048** | 0.064 |
| bone010 | † | 0.700 | **0.266** | † | 383.214 | **123.943** | † | 41.750 | **31.119** | † | **1.956** | 2.000 | † | **0.047** | 0.064 |
| ldoor | † | 0.106 | **0.055** | † | 81.043 | **32.788** | † | 18.467 | **15.515** | † | **1.889** | 2.012 | † | **0.101** | 0.126 |
| boneS10 | † | 0.363 | **0.180** | † | 175.872 | **65.295** | † | 27.220 | **22.737** | † | **2.138** | 2.257 | † | **0.078** | 0.099 |
| Emilia_923 | † | † | **0.309** | † | † | **134.476** | † | † | **38.669** | † | † | **2.085** | † | † | **0.054** |
| inline_1 | † | 0.673 | **0.263** | † | 322.954 | **104.816** | † | 34.164 | **26.351** | † | **1.804** | 2.045 | † | **0.052** | 0.077 |
| Fault_639 | † | † | **0.315** | † | † | **125.997** | † | † | **37.617** | † | † | **1.773** | † | † | **0.045** |
| StocF-1465 | † | 0.607 | **0.227** | † | 332.244 | **110.737** | † | 41.615 | **28.841** | † | **1.661** | 1.731 | † | **0.039** | 0.059 |
| msdoor | † | 0.128 | **0.066** | † | 88.255 | **36.339** | † | 18.708 | **14.662** | † | **1.618** | 1.776 | † | **0.086** | 0.121 |
| af_5_k101 | **0.021** | 0.082 | 0.040 | **16.148** | 68.694 | 27.598 | **2.624** | 13.885 | 12.210 | 2.624 | **1.733** | 1.788 | 1.000 | **0.123** | 0.146 |

**Table 4: Per parallel step results of Distributed Southwell (DS) compared with Parallel Southwell (PS) and Block Jacobi (BJ) for taking 50 parallel steps using 8192 MPI processes. Mean wall-clock time and communication cost over the 50 parallel steps are shown.**

| | Wall-clock time | | | Communication cost | | |
|---|---|---|---|---|---|---|
| Matrix | BJ | PS | DS | BJ | PS | DS |
| Flan_1565 | 0.017 | 0.012 | **0.006** | 12.537 | 7.307 | **3.056** |
| audikw_1 | 0.031 | 0.025 | **0.013** | 18.092 | 10.634 | **5.204** |
| Serena | 0.023 | 0.017 | **0.009** | 15.234 | 8.899 | **3.607** |
| Geo_1438 | 0.018 | 0.013 | **0.007** | 14.149 | 7.854 | **3.337** |
| Hook_1498 | 0.021 | 0.014 | **0.008** | 13.599 | 8.102 | **3.705** |
| bone010 | 0.022 | 0.017 | **0.007** | 14.596 | 9.024 | **3.406** |
| ldoor | 0.008 | 0.006 | **0.004** | 6.319 | 4.243 | **2.688** |
| boneS10 | 0.018 | 0.014 | **0.006** | 9.226 | 6.026 | **2.562** |
| Emilia_923 | 0.023 | 0.014 | **0.008** | 15.370 | 7.574 | **3.473** |
| inline_1 | 0.025 | 0.019 | **0.010** | 13.877 | 9.191 | **5.075** |
| Fault_639 | 0.021 | 0.015 | **0.008** | 15.735 | 7.317 | **3.411** |
| StocF-1465 | 0.021 | 0.014 | **0.009** | 14.616 | 7.798 | **4.455** |
| msdoor | 0.009 | 0.007 | **0.005** | 7.101 | 4.467 | **2.955** |
| af_5_k101 | 0.008 | 0.006 | **0.004** | 6.155 | 4.728 | **3.248** |

the two wall-clock times are quite close. Additionally, when Block Jacobi achieves $\|r\|_2 = 0.1$, it is faster than Parallel and Distributed Southwell, e.g., for Hook_1498. However, it is often the case that Block Jacobi cannot achieve $\|r\|_2 = 0.1$, even for a small number of processes. For example, for Flan_1565, ldoor, and StocF-1465, Block Jacobi cannot achieve $\|r\|_2 = 0.1$ for more than 128 processes. This demonstrates that Block Jacobi can be an unreliable method even for a small number of processes.

We now look at the residual norm after 50 parallel steps of each method as we vary the number of MPI processes from 32 to 8192 (from 1 to 256 nodes). Six examples are shown in Figure 9. It is clear that for larger numbers of MPI processes, the convergence of Block Jacobi severely degrades or Block Jacobi may even diverge after 50 parallel steps. The degradation is much more mild for Parallel Southwell and Distributed Southwell. The fact that the residual norm of Distributed Southwell does not significantly degrade is why it can be considered a competitor to Block Jacobi for massively parallel multigrid smoothing and preconditioning.

## 5 RELATED WORK

Several variants of Southwell's original method have been reported in the literature that are designed to reduce the cost of choosing the next row to relax and/or allow more than one equation to be relaxed at the same time.

In the *sequential adaptive relaxation* method [14, 13], a small active set of rows is initially chosen. A row from this active set is chosen based on its residual and a preliminary relaxation is performed. If the updated value is not a significant change from the previous value, then the update is discarded and the row is removed from the active set. Otherwise, the updated value is kept, and the neighbors of the row are added to the active set. The number of rows to consider in each step is thus kept small in this strategy.

Alternatively, in the *simultaneous adaptive relaxation* method [14], a threshold $\theta$ is chosen. Rows with residual components larger than $\theta$ in magnitude are relaxed simultaneously. We note that such methods, like Jacobi, are not guaranteed to converge for all symmetric positive definite matrices, whereas such convergence is guaranteed for Multicolor Gauss-Seidel and Parallel Southwell, where an independent set of equations is relaxed simultaneously. Stagewise orthogonal matching pursuit methods also are accelerated by using the idea of a threshold to select multiple basis vectors simultaneously [5, 9].

In the context of large-scale optimization, greedy coordinate descent has been parallelized by partitioning the problem into subdomains. Each subdomain is solved using the greedy method corresponding to Sequential Southwell [19].

To reduce the number of messages sent and to improve the efficiency of an asynchronous iterative method, an *asynchronous variable threshold* method has been designed [8]. Here, thresholds are applied to the change in the solution after a block of equations

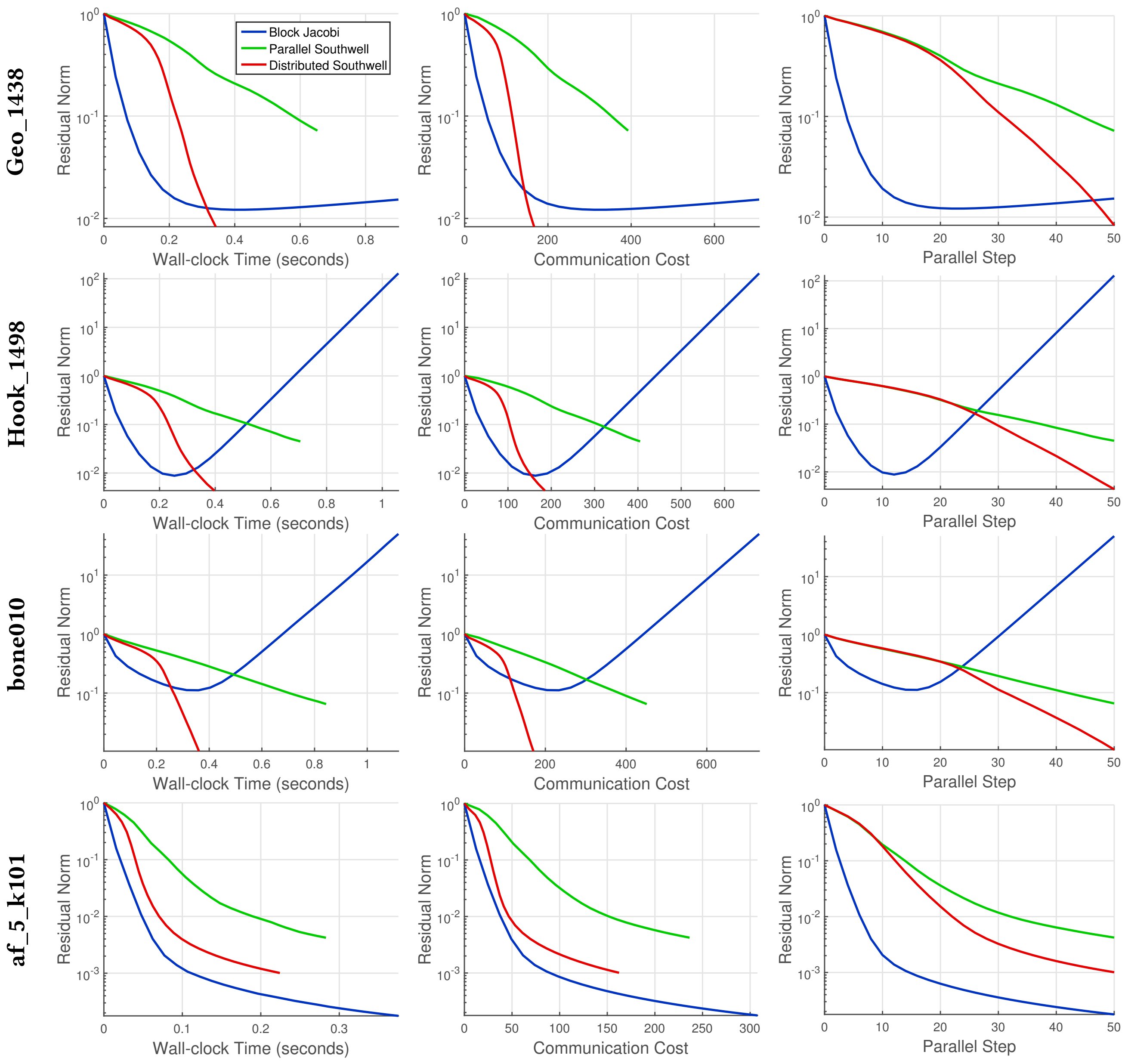


**Figure 7: Comparison of Block Jacobi and Distributed and Parallel Southwell for four test problems that show different behavior of Block Jacobi. For Geo_1438 and Hook_1498, Block Jacobi is able to reach the target residual norm of 0.1, and is the best method for these problems for this level of accuracy. However, Block Jacobi diverges for these problems if more steps are taken. For bone010, Block Jacobi is not able to reach the target residual norm of 0.1. Distributed Southwell is the best method for this problem for this level of accuracy. Of the 14 test problems shown in Table 1, af_5_k101 is the only case in which Block Jacobi never diverged.**

corresponding to a subdomain or process has been relaxed (like in the sequential adaptive relaxation method mentioned above). If the change is too small, the update is not performed, and thus no messages need to be sent in this case. This method is not related to Distributed Southwell, but presents a possibility for further reducing communication cost.

Southwell-based techniques have been used by Rüde [14, 13] as adaptive smoothers for problems with irregular geometries or jumps in coefficients where there may be locally large residuals in the multigrid method. Sequential adaptive relaxation and simultaneous adaptive relaxation, mentioned above, were applied to augment a standard smoothing step. For the simple problem we tested in Section 4.1, on the other hand, we found that even a “1/2 sweep” of Distributed Southwell could give grid-independent convergence.

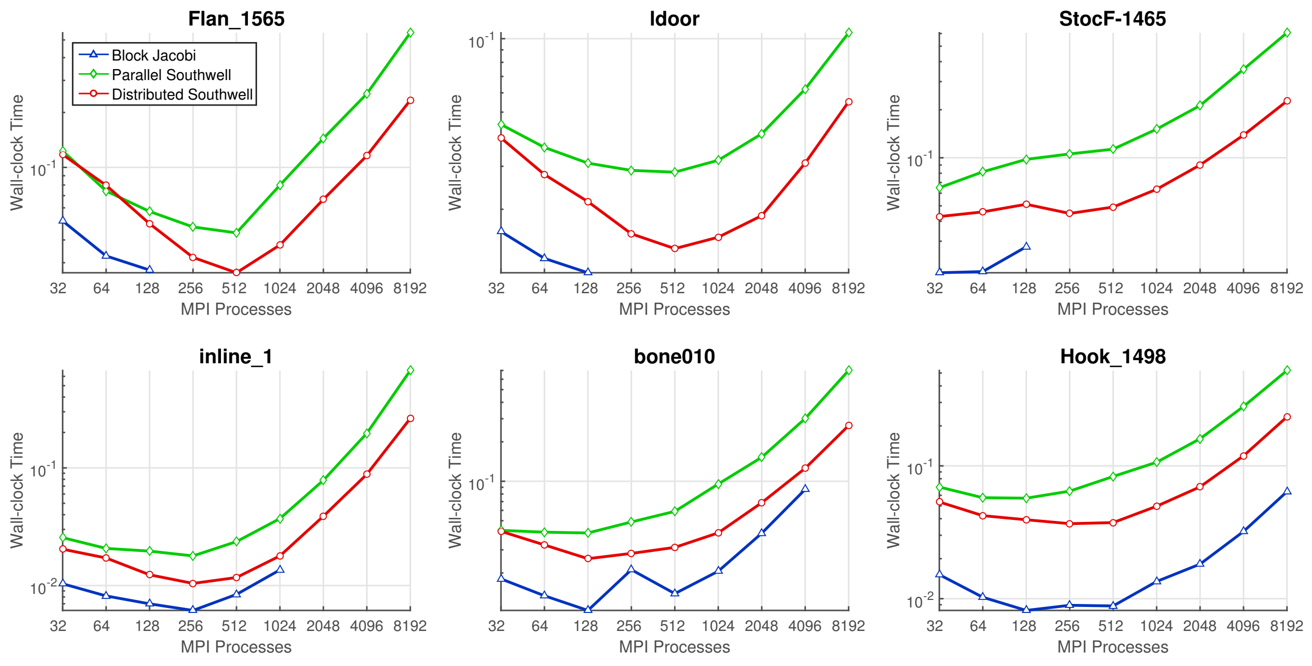


Figure 8: Wall-clock time as a function of the number of MPI processes for reducing $\|r\|_2$ to 0.1. Missing data for Block Jacobi indicates that Block Jacobi could not achieve $\|r\|_2 \leq .1$ in 50 parallel steps, usually due to divergence of the Block Jacobi method. The Block Jacobi method is fastest when it does converge.

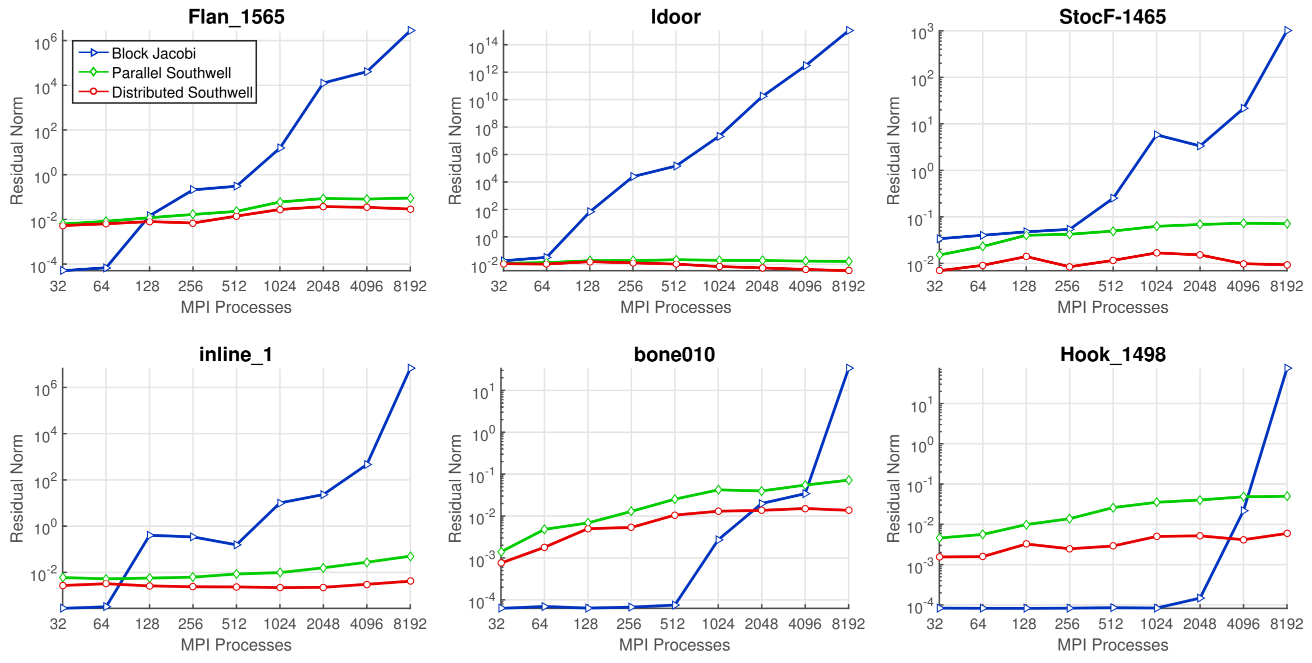


Figure 9: Residual norm after 50 parallel steps as a function of the number of MPI processes for different test problems. When the residual norm is above 1, this indicates that the method has diverged after 50 parallel steps. For larger numbers of processes, Block Jacobi is more likely to diverge after many steps.

## 6 CONCLUSION

Parallel Southwell is a natural way to parallelize the Sequential Southwell method. However, in a distributed setting, Parallel Southwell has a high communication cost that stems from the requirement for neighboring processes to exchange the residual norms of their local subproblems. In Distributed Southwell, the main idea is that these residual norms do not need to be known exactly. Instead, estimates of the residual norms of neighbors can be computed locally without communication. However, deadlock may occur if the estimates are such that no process thinks it has the largest residual norm. This paper presented a novel scheme to avoid deadlock by sending explicit residual norm update messages *only when necessary*. The result is that Distributed Southwell uses much less communication than Parallel Southwell.

Distributed Southwell is also a potential improvement over Block Jacobi when a large number of processes is used. This is an important issue when considering future exascale machines, where the number of cores will be massive. In this case, block sizes will be small, possibly leading to slow or no convergence for Block Jacobi.

## 7 ACKNOWLEDGMENTS

The authors are grateful to the reviewers whose perceptive and detailed comments greatly helped improve this paper. 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. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

## REFERENCES

[1] *MPI: Message Passing Interface Standard, Version 3.0*, High-Performance Computing Center Stuttgart, September 2012.

[2] M. Adams, M. Brezina, J. Hu, and R. S. Tuminaro, *Parallel multigrid smoothing: polynomial versus Gauss-Seidel*, Journal of Computational Physics, 188 (2003), pp. 593–610.

[3] M. F. Adams, *A distributed memory unstructured Gauss-Seidel algorithm for multigrid smoothers*, in Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, 2001.

[4] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, *Multigrid smoothers for ultraparallel computing*, SIAM Journal on Scientific Computing, 33 (2011), pp. 2864–2887.

[5] T. Blumensath and M. E. Davies, *Stagewise weak gradient pursuits*, IEEE Transactions on Signal Processing, 57 (2009), pp. 4333–4346.

[6] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, , and U. M. Yang, *A survey of parallelization techniques for multigrid solvers*, Frontiers of Parallel Processing for Scientific Computing, (2005).

[7] T. Davis and Y. Hu, *The University of Florida sparse matrix collection*, ACM Transactions on Mathematical Software, 38 (2011), pp. 1:1–1:25.

[8] D. de Jager and J. Bradley, *Extracting state-based performance metrics using asynchronous iterative techniques*, Performance Evaluation, 67 (2010), pp. 1353–1372.

[9] D. L. Donoho, Y. Tsaig, I. Drori, and J.-L. Starck, *Sparse solution of underdetermined systems of linear equations by stagewise orthogonal matching pursuit*, IEEE Transactions on Information Theory, 58 (2012), pp. 1094–1121.

[10] M. Griebel and P. Oswald, *Greedy and randomized versions of the multiplicative Schwarz method*, Linear Algebra and its Applications, 437 (2012), pp. 1596–1610.

[11] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on Scientific Computing, 20 (1998), pp. 359–392.

[12] J. Nutini, M. Schmidt, I. Laradji, M. Friedlander, and H. Koepke, *Coordinate descent converges faster with the Gauss-Southwell rule than random selection*, in ICML-15 Proceedings of the 32nd International Conference on Machine Learning, 2015, pp. 1632–1641.

[13] U. Rüde, *Fully adaptive multigrid methods*, SIAM Journal on Numerical Analysis, 30 (1993), pp. 230–248.

[14] U. Rüde, *Mathematical and Computational Techniques for Multilevel Adaptive Methods*, SIAM, Philadelphia, PA, USA, 1993.

[15] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, PA, USA, 2nd ed., 2003.

[16] R. V. Southwell, *Relaxation Methods in Engineering Science – A Treatise on Approximate Computation*, Oxford University Press, 1940.

[17] R. V. Southwell, *Relaxation Methods in Theoretical Physics*, Clarendon Press, 1946.

[18] J. Wolfson-Pou and E. Chow, *Reducing communication in distributed asynchronous iterative methods*, in ICCS Workshop on Mathematical Methods and Algorithms for Extreme Scale (Procedia Computer Science), vol. 80, 2016, pp. 1906–1916.

[19] Y. You, X. Lian, J. Liu, H. Yu, I. S. Dhillon, J. Demmel, and C. Hsieh, *Asynchronous parallel greedy coordinate descent*, in Advances in Neural Information Processing Systems 29, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, eds., Curran Associates, Inc., 2016, pp. 4682–4690.

# A ARTIFACT DESCRIPTION: *DISTRIBUTED SOUTHWELL: AN ITERATIVE METHOD WITH LOW COMMUNICATION COSTS*

## A.1 Abstract

This artifact is comprised of the code for Distributed Southwell (and the other methods used in this paper), the SuiteSparse test matrices from Table 1, and files containing the random initial guesses for $x$ in the matrix equation $Ax = b$. It also includes Bash scripts needed to reproduce the results shown in this paper.

## A.2 Description

*A.2.1 Check-list (artifact meta information).*

- **Algorithm:** Distributed Southwell and Parallel Southwell
- **Program:** `C++` with `MPI`.
- **Compilation:** `CC -std=c++11 -O3 -qopenmp -mkl` on Cori. If not on Cori, use Intel MPI instead, i.e., use `mpiicpc` instead of `CC`.
- **Binary:** `DMEM_Southwell`.
- **Data set:** binary files containing SuiteSparse matrices and text files containing initial $x$ vectors.
- **Run-time environment:** Linux x86_64, Cray MPICH 7.3.0, Casper, METIS, and Intel Math Kernel Library (any version that supports PARDISO and random number generators can be used).
- **Hardware:** Any.
- **Output:** Residual norm and statistics for setup and solve phases, e.g., total wall-clock time.
- **Experiment workflow:** Clone the repository; install libraries (METIS and Casper) if necessary; compile with `make`; run `DMEM_Southwell` with desired input.
- **Publicly available?:** Yes.

*A.2.2 How software can be obtained.* It can be obtained from Github by cloning the repository `https://github.com/jwolfsonp/Southwell.git`.

*A.2.3 Hardware dependencies.* We gathered our results using nodes on the NERSC Cori (Phase I), where each node contains two Intel Xeon E5-2698 v3 Haswell 2.3 GHz CPUs and are connected via Cray Aries interconnect. However, because Casper is used, our program should run on most machines.

*A.2.4 Software dependencies.*

- **`C++11 with OpenMP`**.
- **Cray MPICH:** We used Cray MPICH on Cori with the default compiler `CC`. If not running on Cori, use `mpiicpc`, i.e., the latest Intel C++ MPI.
- **Casper**: We used Casper for one-sided asynchronous progress control. The user can specify a certain number of ghost processes per node by setting the environment variable `CSP_NG`, e.g., executing `export CSP_NG=1` on the command line will set aside one physical core per node for asynchronous progress control. For the results in this paper, only one core was used. The static library file is included in the repository, which was generated on Cori.
- **METIS:** This is needed for load balancing. The static library file is included in the repository, which was generated on Cori.
- **Intel Math Kernel Library (MKL):** For solving local subdomains directly, PARDISO is used. We also need MKL's random number generators for generating initial guesses. MKL is not needed to reproduce the results in this paper, but the option to use PARDISO and random number generators is included in case users are interested.

*A.2.5 Datasets.* The SuiteSparse matrices are stored in a Dropbox folder (provided upon request by emailing Jordi Wolfson-Pou at jwp3@gatech.edu). The text files containing initial guesses are stored on Github.

## A.3 Installation

Simply clone the repository, and use `make`. If using Cori, type

```
module unload cray-libsci
module unload darshan
module unload cray-mpich
module load cray-mpich/7.3.1
```

before using `make`. If not running on Cori, be aware of the following.

1. When compiling, make sure MKL is linked correctly.
2. Make sure METIS and Casper are correctly installed.
3. the Makefile is currently only suitable for Cori, so it must be manually altered to run on a different machine.

## A.4 Experiment workflow

After compiling, the `DMEM_Southwell` binary should be in your directory. The program takes arguments as input in the format `-argument argument_value`. Some arguments just set flags, so no argument value is needed.

Here is an example of running our program at the command line (Note that the arguments at every new line are there just for readability).

```
CSP_NG=1 srun -N 32 -n 1024 ./DMEM_Southwell
        -x_zeros
        -mat_file ecology2.mtx.bin
        -sweep_max 20
        -loc_solver gs
        -solver sos_sds
```

In this example, 32 nodes of Cori are requested, with one ghost process per node. An initial guess of all zeros is used for $x$ and $b$ is initialized to uniformly distributed random numbers. For Cori, `srun` is used, where `-N` specifies number of nodes, and `-n` is the total number of MPI processes. If using a different machine, `mpirun` or `mpiexec` can be used. Our program automatically scales $x$ or $b$ (depending on if $b$ or $x$ are requested by the user to be a vector of all zeros) such that the initial residual has a norm of one. In this case, by specifying `x_zeros`, $x$ is all zeros, and $b$ is scaled. Setting $b$ to all zeros is the default. The matrix ecology2 is used here, where `-mat_file` expects a file name as the argument value. If a file is not specified, our program generates a 5-point centered-difference approximation of the Laplace PDE on a $1000 \times 1000$ 2D domain. We run 20 parallel steps, as specified by the `sweep_max 20` argument, which is also the default. We use Gauss-Seidel sweeps on local subdomains, as denoted by `-loc_solver gs`, which is the default. For the parallel solver, we use Distributed Southwell as specified by `-solver sos_sds`. No solver is used by default.

During execution, the matrix is loaded to MPI rank 0, and partitioned using METIS. The matrix is then scattered to the rest of the processes. Each process then gathers necessary information for the solve phase, e.g., determines all process neighbors. The timings for the setup phase is then written to the screen. After the desired number of parallel steps is carried out by the desired solver, various statistics are reported to the screen, e.g., residual norm upon completion. To print the data in a format suitable for post-processing, e.g., for easily plotting the output data, the `-format_out` argument can be used.

## A.5 Evaluation and expected result

To reproduce the parallel data presented in this paper, the script `AllMatJob.sh` runs the `SouthwellJob.sh` script on all 14 matrices from Table 1. The `SouthwellJob.sh` script submits a parallel job on Cori and runs the `SweepPar.sh` script. The user must manual change the number of nodes to request within the `SouthwellJob.sh` script.

The `SweepPar.sh` script executes 50 parallel steps of Distributed Southwell, Parallel Southwell, and Block Jacobi, with Gauss-Seidel sweeps on local subdomains. The script uses 32 MPI processes per node. A random $x$ read from a file is used. The script takes the matrix name as the first input (the file will be `matrix_name.mtx.bin`), and the number of nodes second. For example, `./SweepPar.sh Flan_1565 32` loads the file `Flan_1565.mtx.bin` and uses 32 nodes with 32 processes per node. The script produces text files stored in the `data/matrix_name/` directory, one file per method per number of nodes.