



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Considerations on the implementation and use of Anderson acceleration on distributed memory and GPU-based parallel computers

J. Loffeld, C. S. Woodward

August 5, 2015

Association for Women in Mathematics Research Symposium
Baltimore, MD, United States
April 10, 2015 through April 11, 2015

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Considerations on the implementation and use of Anderson acceleration on distributed memory and GPU-based parallel computers

John Loffeld and Carol S. Woodward
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
loffeld1@llnl.gov; woodward6@llnl.gov

Abstract Recent work suggests that Anderson acceleration can be used as an accelerator to the fixed-point iterative method. To improve the viability of the algorithm, we seek to improve its computational efficiency on parallel machines. The primary kernel of the method is a least-squares minimization within the main loop. We consider two approaches to reducing its cost. The first is to use a communication-avoiding QR factorization, and the second is to employ a GMRES-like restarting procedure. On problems using 1,000 processors or less, we find the amount of communication too low to justify communication avoidance. The restarting procedure also proves not to be better than current approaches unless the cost of the function evaluation is very small. In order to begin taking advantage of current trends in machine architecture, we also studied a first-attempt single-node GPU implementation of Anderson acceleration. Performance results show that for sufficiently large problems a GPU implementation can provide a significant performance increase over CPU versions due to the GPU's higher memory bandwidth.

Keywords: Anderson acceleration, nonlinear solvers, fixed-point iteration, TSQR

AMS subject classifications: 65B99, 65N22, 65H10

1 Introduction

Nonlinear root finding problems of the form $f(u) = 0$ are common in computational science problems and especially when computing the solution of discretized PDEs. For large-scale systems, the Newton-Krylov method is commonly used due to the fast, often quadratic, convergence of the Newton iteration [4, 12] and the scalability of linear Krylov methods. However, the need to solve a large linear system each iteration involving the Jacobian of f results in high computational cost per step. Furthermore, for complex problems, finding an analytical expression for the Jaco-

bian can be non-trivial or evaluation of the Jacobian may be expensive (see e.g., [10]). Numerical approximations of the action of the Jacobian times a vector can be used instead, but doing so can compromise the rate of convergence of the Krylov iteration. In either case, a scalable preconditioner is often required for good performance when solving the linear systems, and the preconditioner adds considerable complexity to the solution process.

The nonlinear problem can be posed as a stationary problem $u = g(u)$ through the relation $f(u) = u - g(u) = 0$. Fixed-point iteration can then be applied to this system. Compared to methods based on the Newton iteration, the fixed-point method is simple to implement and has a lower computational cost per step, as it does not require use of the derivative. Unfortunately, the iteration does not always converge since the function, $g(u)$, must be a contraction map [11]. When the iteration does converge, the rate of convergence is often slow, typically only linear. However, recent work has shown that the rate of convergence of fixed-point iteration can be improved through Anderson acceleration (AA) [16, 2]. This work raises the possibility of using Anderson-accelerated fixed-point iteration as an alternative to Newton-Krylov in cases where determining the Jacobian is difficult, evaluating the Jacobian is expensive, or the rate of convergence of the Krylov iteration is slow due to a lack of a good preconditioner.

AA improves the rate of convergence of the fixed-point iteration by utilizing information from more than just the most previous iterate. For each iterate, it chooses a linear combination over m prior iterates that minimizes the fixed-point residual in the least-squares sense. This approach of maximizing rate of convergence through a residual minimizing choice of next iterate is similar to the idea behind the generalized minimum residual method (GMRES) iterative linear solver. Indeed, on linear problems, a mathematical equivalence in the rate of convergence between AA and GMRES has been shown in [16]. The size of the least-squares problem that must be solved each AA iteration is $n \times m$, where n is the number of unknowns in the problem. Solving a large least-squares problem each iteration makes AA more expensive per step than basic fixed-point iteration. If $g(u)$ is not a dominant cost, minimizing the cost of the least-squares problem is the key to making the method efficient. In this paper, we consider two approaches to implementing AA that are aimed at lowering the cost of solving the least squares problem. Both cases compromise the rate of convergence of the iteration, so any net benefit from either approach is a matter of balancing cost per iteration and the number of iterations that must be computed. We found the trade-off in one case not to be favorable for the sizes of problems we tested, but the other approach to be modestly favorable in some instances.

The first approach is aimed at large-scale problems implemented on distributed-memory machines with at least thousands of processors and problem sizes with tens of thousands or more unknowns per processor. We consider whether it is possible, or even necessary, to lower the MPI communication cost of the least squares problem. In Anderson acceleration, the least-squares problem can be solved in a variety of ways. QR factorization is a good choice due to a balance between efficiency and accuracy [16, 7]. With some approaches for the QR problem, the factorization can be incrementally updated each iteration without requiring a full factorization [8, 15].

When computing the QR decomposition on distributed-memory machines, use of panel factorization, blocking and tiling optimizations, such as those employed in the ScaLAPACK library [3, 5], can give a significant performance benefit over unoptimized algorithms. Recent communication-avoiding QR algorithms use such techniques to minimize interprocessor communication cost, which may be of particular benefit on large-scale machines [6]. Such algorithms minimize communication of the same matrix elements by performing operations on sub-matrices of size greater than rank one. Such techniques are generally unsuitable for per-iteration update of the factorization. Employing those algorithms in the context of AA would require updating the QR factorization only every k vectors, limiting the degree of acceleration in the fixed-point iteration. Nevertheless, use of such algorithms might be beneficial if the computational savings outweigh the cost of computing additional iterations.

The second approach, restarting, is applicable to all sizes of problems, as well as both serial and parallel implementations. In this paper, we tested it on small-scale parallel problems. Restarting is a commonly used technique for reducing cost in GMRES [13]. For linear problems, a mathematical equivalence in the rate of convergence between a truncated-and-restarted AA and a restarted GMRES has been shown [16]. Despite the equivalence, the underlying operations computed by each method are different, i.e. they have a different computational cost structure. However, costs are comparable enough between them that one would expect the trade-off between rate of convergence and computational savings to balance similarly between the two methods on linear problems. However, in AA, rather than periodically truncating the iteration and restarting with the last iterate, a factorization over a sliding window of k previous iterates can be maintained [16]. Thus, rather than periodically fully discarding all previous iterates and starting over with a single vector, only the most stale iterate is deleted each iteration. This contrasting approach, which we refer to as “sliding”, allows a less severe reduction in the rate of convergence, at the cost of applying the delete procedure each iteration. We tested the overhead to determine whether its additional cost was worth the better acceleration. We found that avoiding the computational cost of the delete operation through restarting is modestly beneficial in some cases but not all.

Finally, we tested the performance of a GPU implementation of Anderson acceleration as a whole. High Performance Computing machines increasingly employ accelerators such as GPUs, due to their high concurrency. For an algorithm to be well-suited for current and future machines, a significant amount of its parallelism must be captured within each node through the accelerator. Parallelism through MPI alone is no longer sufficient. As a step towards an implementation of Anderson acceleration suitable for current and future supercomputers, we developed a single-node GPU implementation of Anderson acceleration that is based on the GPU-optimized BLAS library, CuBLAS. GPUs are balanced differently than CPUs and require a higher degree of concurrency to operate efficiently. On the machines we tested, we found that the GPU version was quite inferior to the CPU version when the number of unknowns in a problem was about 10,000 or less. When the number was greater,

the GPU version was able to greatly outperform the CPU version due to the higher memory bandwidth on the device.

This paper is organized as follows. Section 2 describes the MPI-based implementation of AA used for the experiments. Section 3 describes some performance measurements of the implementation to determine the balance of local computation and interprocessor communication. Section 4 details performance comparisons between a communication avoiding implementation of AA versus the base implementation. In Section 5 we give a performance comparison of a restarted version of AA versus the base implementation, which uses a sliding window of past iterates. We describe and give performance results for the GPU implementation in Section 6. Lastly, in Section 7, we make some final conclusions and describe some possible future work.

2 Anderson Acceleration

In this section we describe the baseline implementation of AA used in our numerical experiments. The implementation is part of the C language KINSOL package of solvers for nonlinear algebraic equations from the SUNDIALS suite of codes [9, 1]. Methods in SUNDIALS are written on top of an abstracted vector API so that they are independent of whether and how parallelism is used. The SUNDIALS distribution is equipped with a number of packages that include implementations of the vector kernels for serial, thread-parallel, and distributed memory parallel (with MPI) vectors, although users can supply their own. The library abstracts away details about how the data is mapped on to processors and how communication is handled between processors when computing operations on the vectors. As such, we specify the implementation of AA below in a “Matlab-like” manner, only specifying details about parallelization when needed.

Our goal is to solve fixed-point problems of the form: Given $g : \mathbb{R}^n \rightarrow \mathbb{R}^n$, find u such that $u = g(u)$. The Anderson accelerated fixed-point method is given in Algorithm 1.

Algorithm 1: Anderson acceleration

Input: u_0 , $m \geq 1$, and ε

$u_1 \leftarrow g(u_0)$

for $i = 1, 2, \dots$, *until* $\|u_{i+1} - u_i\| < \varepsilon$ **do**

$m_i \leftarrow \min\{m, i\}$

$H_i \leftarrow [f_{i-m_i}, \dots, f_i]$, where $f_j = g(u_j) - u_j$

Solve the constrained least-squares problem for $\alpha^{(i)} = (\alpha_0^{(i)}, \dots, \alpha_{m_i}^{(i)})^T$ s.t.

$\min_{\alpha} \|H_i \alpha\|_2$ s.t. $\sum_{j=0}^{m_i} \alpha_j = 1$

$u_{i+1} \leftarrow \sum_{j=0}^{m_i} \alpha_j^{(i)} g(u_{i-m_i+j})$

end

Output u_i

In practical implementation, the constrained least-squares problem is often formulated as the following equivalent unconstrained least-squares problem [7, 16]: find $\gamma^{(i)} = (\gamma_0^{(i)}, \dots, \gamma_{m_i-1}^{(i)})^T$ such that $\min_{\gamma} \|f_i - F_i \gamma\|_2$, where $F_i \equiv [\Delta f_{i-m_i}, \dots, \Delta f_{i-1}]$ and $\Delta f_j = f_{j+1} - f_j$. The least-squares coefficient vectors α and γ are related by $\alpha_0 = \gamma_0$, $\alpha_j = \gamma_j - \gamma_{j-1}$ for $1 \leq j \leq m_i - 1$, and $\alpha_{m_i} = 1 - \gamma_{m_i-1}$. The next iterate then becomes $u_{i+1} = g(u_i) - \sum_{j=0}^{m_i-1} \gamma_j^{(i)} [g(u_{i-m_i+j+1}) - g(u_{i-m_i+j})]$.

The KINSOL implementation of AA follows the approach described by Walker in [15]. The least-squares problem is solved by performing the QR factorization of F_i and using backward substitution to solve the upper triangular system $R_i \gamma = Q_i^T f_i$. Note that when $i < m$, the size of F is $n \times i$, and a new vector is added to the right of F_i in each iteration. After the m -th iteration, F_i remains of fixed size, $n \times m$, but in each iteration a column vector is removed from the left of F_i while a new vector is added to the right. It is inefficient to re-factorize F_i anew each step, so two helper procedures are used to in-place update Q_i and R_i based on the previous factorization. QRAdd updates the factorization to account for the addition of a vector to the right of F_i , while QRDelete updates for the removal of a vector from the left.

QRAdd uses modified Gram-Schmidt to orthonormalize each new Δf_{i-1} against the previous columns of Q_{i-1} . The resulting vector becomes the new rightmost column of Q_i . Algorithm 2 gives pseudo-code for the procedure using Matlab notation.

Algorithm 2: QRAdd

Input : $Q \in \mathbb{R}^{n \times m_i}$, $R \in \mathbb{R}^{m_i \times m_i}$, and Δf_{i-1}
Output: $Q \in \mathbb{R}^{n \times m_i+1}$ and $R \in \mathbb{R}^{m_i+1 \times m_i+1}$

for $j = 1$ to $m_i - 1$ **do**
 $R(j, m) \leftarrow Q(:, j)^T * \Delta f_{m-1}$
 $\Delta f_{i-1} \leftarrow \Delta f_{i-1} - R(j, m) * Q(:, j)$
end
 $Q(:, m) \leftarrow \Delta f_{i-1} / \|\Delta f_{i-1}\|_2$ and $R(m, m) \leftarrow \|\Delta f_{i-1}\|_2$.

On a distributed-memory machine with p processors, Q_i is represented as a set of column vectors, with each processor receiving n/p rows of each vector. Communication between processors is incurred by the dot products $Q(:, j)^T * \Delta f_{m-1}$ and when computing the norm $\|\Delta f_{i-1}\|_2$ (implemented with a dot product). The results of the dot products are broadcast to all processors, resulting in a copy of R_i on each processor.

QRDelete uses Givens rotations to update the factorization when a vector is removed from F . The procedure is based on the observation that if $F_{k-1} = Q * R$ then $F_{k-1}(:, 2 : m) = Q * R(:, 2 : m)$, where $R(:, 2 : m)$ is upper Hessenberg. Note that Q and $R(:, 2 : m)$ do not constitute a QR factorization of $F_{k-1}(:, 2 : m)$. They can be updated to be one by using Givens rotations to return $R(:, 2 : m)$ to upper triangular form and then applying the inverse of those rotations to Q . Specifically, if we determine Givens rotations J_1, \dots, J_{m-1} such that $J_{m-1} * \dots * J_1 * R(:, 2 : m)$ is upper triangular, then

$$F_{k-1}(:, 2:m) = Q * R(:, 2:m) = Q * J_1' * \dots * J_{m-1}' * J_{m-1} * \dots * J_1 * R(:, 2:m),$$

and setting $Q = Q * J_1' * \dots * J_{m-1}'$ and $R = J_{m-1} * \dots * J_1 * R(:, 2:m)$ gives a QR factorization for F_{k-1} . The pseudo-code for QRDelete is shown in Algorithm 3.

Algorithm 3: QRDelete

Input : $Q \in \mathbb{R}^{n \times m}$ and $R \in \mathbb{R}^{m \times m}$

Output: $Q \in \mathbb{R}^{n \times m-1}$ and $R \in \mathbb{R}^{m-1 \times m-1}$

for $i = 1$ to $m - 1$ **do**

$$b \leftarrow \sqrt{R(i, i+1)^2 + R(i+1, i+1)^2}$$

$$c \leftarrow R(i, i+1)/b \text{ and } s \leftarrow R(i+1, i+1)/b$$

$$R(i, i+1) \leftarrow d \text{ and } R(i+1, i+1) \leftarrow 0$$

if $i < m - 1$ **then**

for $j = i + 2$ to m **do**

$$d \leftarrow c * R(i, j) + s * R(i+1, j)$$

$$R(i+1, j) \leftarrow -s * R(i, j) + c * R(i+1, j) \text{ and } R(i, j) \leftarrow d$$

end

end

$$V \leftarrow c * Q(:, i) + s * Q(:, i+1)$$

$$Q(:, i+1) \leftarrow -s * Q(:, i) + c * Q(:, i+1) \text{ and } Q(:, i) \leftarrow V$$

end

$$Q \leftarrow Q(:, 1:m-1) \text{ and } R \leftarrow R(1:m-1, 2:m)$$

We are interested in the balance between MPI communication and local on-node cost. The former can be broken down into a bandwidth and latency cost. The latter can be further broken down into the cost of floating-point operations and the cost of data transfers between the processor and memory.

In the case of the SUNDIALS implementation, all MPI communication comes from the dot products found in QRAdd and backwards substitution. On each processor, the dot product kernel sums over the local portion of the vectors itself and only calls MPI's Allreduce routine for the summed value. As a result, the reduction is done only over a single number, and the time spent within MPI is nearly completely a latency cost, with very little bandwidth cost. Only synchronous communication is used, so the processor remains idle during the reduction.

The three main kernels in AA, QRAdd, QRDelete, and backwards substitution, are all comprised solely of vector-vector operations. As such, the ratio of floating-point cost to data transfer cost within the kernels (the arithmetic intensity) is very low, well less than one flop per byte. As a result the on-node cost is dominated by the time spent in streaming data between memory and the processor.

Therefore, in AA, the comparison between local-node cost and MPI communication essentially reduces to a balance between local memory transfer cost and the latency costs for the reductions. The TSQR algorithm reduces communication cost by reducing latency on distributed memory machines, so it targets the main form of communication cost found in AA.

Both the on-node and MPI costs of QRAdd and backwards substitution are very similar. Within each AA iteration, QRAdd performs $m_i - 1$ dot products, performs $m_i - 1$ linear sums, and incurs the data transfer costs of streaming over the involved m_i vectors. Backwards substitution must perform m_i dot products to determine $\gamma^{(i)}$ and then perform m_i linear sums to produce the next iterate vector. It also incurs the data transfer costs of operating on those $m_i + 1$ vectors.

QRDelete is not invoked unless $i > m$, in which case $m_i = m$ and the sizes of the matrices remain fixed at $Q \in \mathbb{R}^{n \times m}$ and $R \in \mathbb{R}^{m \times m}$. In that case, the local cost of QRDelete is on par with the other two kernels in that it applies Givens rotations, implemented as a pair of linear sums, to m vectors (as well as to the very small matrix R). No MPI communication is required. If the number of total iterations is considerably greater than m , the local cost of QRDelete is a substantial portion of the overall on-node cost of the whole method. If the history of iterates is flushed every m iterations and the Anderson iteration restarted with $u_{i+1} = g(u_i) - \sum_{j=0}^{m_i-1} \gamma_j^{(i)} (g(u_{i-m_i+j+1}) - g(u_{i-m_i+j}))$ then QRDelete is not needed and the cost per step would be reduced. However, the rate of convergence of the iteration would also be hindered. We test whether it is favorable to make this trade-off in Section 5.

3 Balance of communication versus computation

In this section we discuss some performance measurements of the ratio of inter-processor communication versus local-node computation for KINSOL’s AA implementation. The tests were conducted on two machines: The Blue Gene Q system “Vulcan”, and the Intel Xeon-based system “Cab”. Vulcan is composed of 24,576 16-core PowerPC A2 processors running at 1.6GHz. The processors are connected by a high-speed, low-latency network configured as a 5D torus. IBM’s BG/Q MPI library, based on MPICH2 1.4, was used as the MPI library for the tests on Vulcan. Cab is a cluster of Intel Xeon E5-2670 processors, with two 8-core CPUs on each node in a shared-memory configuration, and a total of 1,296 nodes. The nodes are connected via an InfiniBand QDR network in a two-stage federated fat-tree.

On both Vulcan and Cab, the implementation was instrumented with MPI’s Wtime function. On Vulcan, the timer has sub-nanosecond resolution, whereas on Cab the timer has microsecond resolution. In both cases, timer accuracy was not a limitation. SUNDIALS only uses blocking communication, so the time spent on communication and computation could be measured independently.

On Vulcan, KINSOL was configured to compute a fixed-point problem using AA for 4 iterations, with a window size of $m = 4$, and for 16 iterations with a window size of $m = 16$ on a problem with 1, 10, 100, 1,000, 10,000, and 100,000 unknowns per processor. The problem was run on 100 processors and 1,000 processors, using only a single core per processor to ensure that all MPI communication was done over the network and not through shared memory. The $g(u)$ problem was a dummy function that returned random values for the vector elements. Note that the cost of

AA depends only on the number of iterations computed, the value for m , the number of elements in a vector per processor, and the number of processors, but does not depend on the contents of the solution vector. The choice of $g(u)$ is irrelevant for cost measurement as long as the number of iterations does not change. In our timings, we ignored the cost of computing $g(u)$; the cost measured is for just the Anderson algorithm itself.

The outcome on Vulcan was very similar in all cases so we only show the largest problem with 1,000 processors and with 16 iterations. The results are shown in Table 1a. The rightmost column gives the percentage of overall cost that was spent on MPI communication. While communication dominates when the number of unknowns per processor is small, it becomes negligible for 10,000 unknowns per processor or greater. For most large-scale problems, 10,000 unknowns per processor is quite lean, so the cases of 10,000 unknowns or greater are the most relevant. For problems where the number of processors is 1,000 or less, communication is not a major cost for AA on Vulcan.

On Cab, the problem setup was the same except for two differences. The problem was run on 100 processors and 256 processors, as 256 was the limit of our access. Furthermore, the problem was additionally run with 1,000,000 unknowns to better show how communication falls off in importance relative to computation. As on Vulcan, the outcomes were quite similar regardless of the number of processors or iterations, so we display only the 16 iteration case with 256 processors in Table 1b. We see that communication on Cab is a greater proportion of overall cost compared to Vulcan, which is not surprising considering the less capable network on Cab. However, the communication cost is still overtaken by the cost of local computation as the number of unknowns increases, and the cost becomes minor even for lean problems.

Overall, we conclude that communication is not a major cost in AA for the scale of problems we have considered. It may be of greater importance when using a larger number of processors.

4 TSQR versus modified Gram-Schmidt

In this section we consider whether the communication-avoiding Tall Skinny QR (TSQR) algorithm might give better performance due to reduced MPI latency compared to the modified Gram-Schmidt implementation in KINSOL (QRAdd). We tested with the distributed-memory TSQR implementation from NuLAB [14]. The details of the algorithm can be found in [6]. We employed the variant where communication is done using a binary reduction tree. The local QR solves on sub-blocks were computed using a LAPACK library optimized for the respective machine, which in the case of Vulcan was IBM's ESSL library and on Cab was Intel's Math Kernel Library (MKL).

The matrices were the same size and used the same partitioning over processors as those in Section 3, allowing us to directly compare the performance of TSQR

Unknowns per processor	Total	Local	MPI	% MPI
1	4.8E-03	4.0E-04	4.4E-03	91.7%
10	4.8E-03	4.6E-04	4.3E-03	90.4%
100	5.7E-03	1.2E-03	4.5E-03	79.0%
1,000	1.4E-02	9.3E-03	5.0E-03	34.9%
10,000	9.3E-02	8.8E-02	5.0E-03	5.4%
100,000	8.8E-01	8.8E-01	5.0E-03	0.6%

(a) Run times on Vulcan

Unknowns per processor	Total	Local	MPI	% MPI
1	6.2E-03	3.4E-05	6.1E-03	99.55%
10	6.1E-03	5.9E-05	6.1E-03	99.0%
100	9.8E-03	6.6E-05	9.7E-03	99.3%
1,000	8.5E-03	3.3E-04	8.1E-03	96.1%
10,000	1.6E-02	3.5E-03	1.3E-02	78.5%
100,000	6.2E-02	4.3E-02	1.9E-02	30.8%
1,000,000	5.6E-01	5.1E-01	4.6E-02	8.2%

(b) Run times on Cab

Table 1 Local node and MPI time costs in seconds for AA in KINSOL when computing for 16 iterations with $m = 16$. On both machines, the MPI cost becomes minor when the number of unknowns per processor is modest or larger.

versus modified Gram-Schmidt. The matrices were filled with random data. Note that the amount of computation and communication does not depend on the content of the matrices, only their dimensions.

The relative performance of TSQR compared to a non-communication-avoiding algorithm improves with the width of the matrix. In particular, TSQR performs better relative to KINSOL for the case when the matrix is 16 columns wide instead of 4 so we only display that case in Table 2. The Vulcan measurements were done on 1,000 processors and the ones on Cab were done using 256 processors.

The far right column of the table shows the overall performance of TSQR relative to modified Gram-Schmidt. We see that on Vulcan the overall cost is significantly reduced. However, the percentage of overall cost that is communication is trivial on Vulcan for problems with even a modest number of unknowns per processor. The cost reduction is from savings in computation not MPI communication. The lower computational cost of TSQR is in part because it uses a tuned library, ESSL, while KINSOL is untuned. For QR factorization, the performance gain of tuned libraries comes primarily through panelization and tiling, which requires operations be performed on sub-matrices wider than a single column. This requirement is at odds with in-place updating the QR factorization one column at a time, as is done in KINSOL and required by the algorithm as written above. To exploit tuned libraries fully, AA would need to perform the QR factorization over k vectors at a time, where the performance gain would increase with k up to some saturation point. That would mean acceleration could be applied only every k -th iteration, and ordinary

fixed-point would need to be used for the iterations in between. To prevent the rate of convergence from being disastrously reduced, k would need to remain small, putting the needs of convergence at odds with the needs of QR algorithm optimizations. It is possible that for some problems there is a balance that results in a net reduction in overall cost, but we have not yet found such a case on Vulcan.

On Cab, the performance of TSQR is unexpectedly poor. As seen in Table 2b, both the communication and computational cost are increased in TSQR over KINSOL by orders of magnitude. We initially believed this was a mistake in our problem setup, but after much investigation we have not found anything particular. The approach taken by TSQR to solving the QR problem is quite different from that of KINSOL, and it appears to balance unfavorably on Cab’s architecture. We will continue to investigate the underlying cause of this. In any case, in light of the results of Section 3, communication avoidance is not expected to be helpful on the scale of problems we have tested.

In conjunction with the tests in Section 3, we conclude that communication avoidance is not generally helpful for problems computed on 1,000 processors or less. For larger scale problems, communication cost may increase relative to computation to the point where avoiding communication becomes important, but that possibility is not tested by our measurements. However, current trends in supercomputer architecture are moving away from large node counts and moving more parallelism to within each node. Current supercomputers such as Sequoia at Lawrence Livermore National Laboratory and Titan at Oak Ridge National Laboratory have large node counts of about 100,000 and 20,000 respectively. The replacement machines are planned to only have several thousand nodes in each case, with most of the parallelism coming from GPUs. This makes the case for use of communication avoidance in Anderson acceleration at the MPI level unconvincing.

5 Restarting

Since communication is not an important cost for the cases we tested, we consider a possible approach for reducing computational cost. As discussed in Section 2, we can restart the iteration every m iterations using only the most recent iterate. As with restarting in GMRES, doing so mitigates the quadratic increase in cost per iteration, but may also reduce the rate of convergence. However, unlike GMRES, AA can also control quadratic growth in cost by limiting the number of past iterates, m , and updating the QR factorization in-place, as discussed in Section 2. This practice is used in the current KINSOL implementation. For easy comparison with “restarting”, we will label this case as “sliding”, since the window of past iterates slides forward each iteration. By limiting m through sliding, the rate of convergence is also reduced, but not as severely as by restarting. The trade-off is that QRDelete must be called each iteration past the m -th one, which increases the cost per iteration.

We tested on Vulcan and Cab whether restarting gives better performance than sliding on a restricted-additive-Schwarz (RAS) iteration applied to the 2D Poisson

Unknowns per processor	Total	Local	MPI	% MPI	Overall % GS
1	2.6E-05	2.2E-05	4.0E-06	15.4%	21.3%
10	3.2E-04	3.2E-04	3.0E-06	0.93%	39.3%
100	1.0E-03	7.1E-04	3.2E-04	31.1%	44.2%
1,000	1.8E-03	1.5E-03	3.2E-04	17.6%	23.0%
10,000	2.8E-02	2.7E-02	3.3E-04	1.2%	32.5%
100,000	6.9E-02	6.8E-02	8.9E-04	1.3%	8.5%

(a) Run times on Vulcan

Unknowns per processor	Total	Local	MPI	% MPI	Overall <i>times</i> GS
1	1.3E-00	6.4E-01	6.6E-01	50.7%	437x
10	2.8E-00	3.9E-01	2.4E-00	85.8%	1045x
100	3.2E-00	3.8E-01	2.8E-00	88.0%	593x
1,000	3.7E-00	8.2E-01	2.9E-00	77.8%	793x
10,000	4.0E-00	8.4E-01	3.2E-00	79.1%	450x
100,000	3.6E-00	1.1E-00	2.5E-00	69.0%	100x

(b) Run times on Cab

Table 2 Local node and MPI time costs (in seconds) for the QR factorization kernel in AA when computed using TSQR, for 16 iterations with a window size of 16. On Vulcan, the communication cost is non-trivially reduced compared with the KINSOL case, but communication is too small a percentage of the overall cost for this to matter. In Cab, the communication cost is actually greatly increased.

problem. Details about the RAS problem can be found in [16], where tests of Anderson acceleration compared to fixed-point iteration.

$$\Delta u + 20u + 20u_x + 20u_y = f \quad \text{in } D = [0, 1] \times [0, 1], \quad \text{where } u = 0 \text{ on } \delta D.$$

The problem was discretized using centered differences discretization on a 128^2 node grid, with $f = -10$. The domain was divided into 4 sub-domains per direction, for a total of 16 sub-domains, with 3 grid lines of overlap between neighbors.

The linear sub-domain problems were solved with a direct solver, but the computational cost of $g(u)$ was ignored. Only the cost of the operations in AA itself were measured. Of course in practice the cost of $g(u)$ may matter greatly, but the complexity varies widely between problems. The measurements are therefore an optimistic bound. Compared to the normal implementation, restarting increases the number of iterations that must be computed, so if the trade-off is not worthwhile when $g(u)$ has zero cost, it will not be worthwhile when the cost of $g(u)$ is included.

Before considering computational efficiency, we first look at how restarting affects convergence compared to sliding. Restarting was compared with sliding on the RAS problem with the following parameters. The problem was run when restarting every 5, 10, and 15 iterations, when sliding with m limited to 5, 10, and 15, and without restriction on m . In what follows, we label that last case the baseline case. For comparison, the problem was also computed using fixed-point iteration with no acceleration. The convergence plots are shown in Figure 1. We see that AA

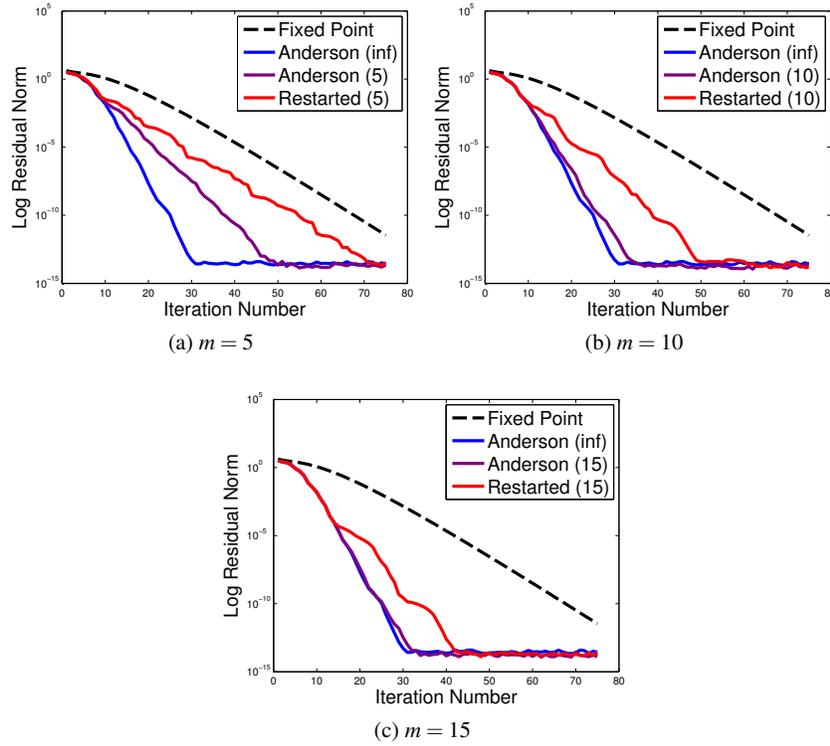


Fig. 1 Comparison of the rate of convergence of restarted Anderson acceleration versus full Anderson acceleration on a 2D Poisson problem. The running times are listed in Table 3.

generally converges much more quickly than the fixed-point iteration. The rate of convergence is reduced for both restarting and sliding, although less so for sliding. As m increases, the rates of convergence improve for both sliding and restarting. When $m = 10$, the rate of convergence for sliding is almost the same as for the baseline case. However, even when $m = 15$, the rate of convergence for restarting is still significantly reduced compared to the baseline case.

We now turn to computational efficiency. On both Vulcan and Cab, restarting was tested using 16 processors using the same parameters as the previous paragraph. The costs, not including that of evaluating $g(u)$, are shown in Tables 3a and 3b for the sizes of m and number of iterations that corresponds to how long it takes to reach a tolerance of 10^{-14} . For example, as can be seen in the figures, it takes about 30 iterations for the baseline iteration to reach the limit of precision. Table 3 shows that it takes 0.072 seconds on Vulcan to compute those 30 iterations. Any case on that machine with lower times to reach machine precision is an improvement on the baseline case.

On Vulcan, we see that sliding and restarting with $m = 5$ give an improvement over the baseline case, and sliding with $m = 10$ gives a negligible improvement. For sliding, even though the cost of QRAdd is reduced due to the smaller window size, QRDelete is also called. We can see this call adds significant overhead. For example, when $m = 15$, sliding needs about the same number of iterations as the baseline case, but the cost of QRAdd is smaller due to the restricted size of m . However, the overall cost is still higher due to the overhead of QRDelete. In contrast, restarting avoids the overhead of QRDelete, but it must compute a larger number of iterations than sliding. The balance results in a cost on Vulcan that is similar between restarting and sliding. When $m = 10$, restarting requires slightly more time than sliding, while when $m = 15$, restarting takes slightly less time.

Restarting is more favorable on Cab, with the most time-consuming case of restarting using less time than the least expensive case of sliding. A window size of $m = 5$ gives the best improvement over the baseline case for both sliding and restarting. In that case, sliding costs about 82% of the baseline case while restarting incurs 62% of the time, which makes restarting 75% of the cost of sliding. The most expensive case for restarting was with $m = 15$, where it incurred 74% of the cost of the baseline case while sliding was 109%.

Type	m	Iters	Time (s)	Type	m	Iters	Time (s)
Sliding	5	47	0.058	Restarted	5	71	0.058
Sliding	10	33	0.071	Restarted	10	49	0.073
Sliding	15	32	0.089	Restarted	15	42	0.085
Baseline	30	30	0.072				

(a) Vulcan

Type	m	Iters	Time (s)	Type	m	Iters	Time (s)
Sliding	5	47	0.028	Restarted	5	71	0.021
Sliding	10	33	0.031	Restarted	10	49	0.022
Sliding	15	32	0.037	Restarted	15	42	0.025
Baseline	30	30	0.034				

(b) Cab

Table 3 Cost of Anderson acceleration when using restarting versus sliding the QR factorization using QRDelete. The number of iterations corresponds to those needed by each iteration in Figure 1. Restarting is less efficient on Vulcan but more efficient on Cab.

We see that the cost of QRDelete is modest to the point that restarting gives no benefit over sliding on Vulcan, even when $g(u)$ has no cost, but still expensive enough that avoiding it gives an improvement on a different machine architecture. This test is only for one problem, and the effect on the rate of convergence of sliding and restarting varies from problem to problem. It can be expected, though, that restarting will require a significantly higher number of iterations over sliding in general. Even on a machine like Cab, the increased number of iterations could be harmful if the cost of $g(u)$ were non-trivial. However, restarting requires almost no additional complexity in the implementation on top of sliding and co-exists with it

easily. Like with most implementations of GMRES, the option can be left to the user and might be valuable on some problems.

6 GPU implementation

On current high performance computing machines, the majority of the compute capacity on each node now comes from accelerators such as GPUs and the Intel Phi line of processors. A well-balanced algorithm for modern supercomputers not only has to be efficient at MPI communication between nodes, but also must make good use of the local accelerators. The architectures of such systems are balanced differently than pure CPU ones, so algorithms must be adapted to take full advantage of them. Along the path for developing an implementation of Anderson acceleration well-suited for modern machines, we have begun work on implementations that make use of accelerators. In this section we describe a first step effort to implement Anderson acceleration on GPUs. The implementation is currently only for a single node and is not yet fully optimized for the GPU architecture, but still shows a considerable performance increase over a CPU implementation. Based on lessons learned from this initial effort, a better optimized and MPI-capable implementation will be developed in future work.

Compared to traditional CPUs, GPUs are characterized by a much higher level of single instruction multiple thread (SIMT) parallelism. For the purposes of this paper, they can be thought of as vector processors, where thousands of vector or matrix elements can be processed in parallel simultaneously using the same instructions. The high SIMT concurrency gives such processors up to an order of magnitude higher peak flops rate than CPUs. GPUs on HPC class machines also have their own RAM, which generally has five to ten times higher bandwidth than the main memory of CPUs. As a trade-off, their caching systems are comparatively limited, and the latency to RAM is also many times higher. Instead of using large low-latency caches to minimize the performance penalty of accessing RAM, as done in CPUs, GPUs instead attempt to hide the latency behind a much higher degree of parallelism. Even if some vector elements are stalled waiting for data to transfer from memory, the high level of concurrency ensures some other elements are likely to have their data requests satisfied and are ready to continue, thus keeping the processor busy.

Algorithms in scientific computing fall within a spectrum between those that are compute bound and those that are memory bound. Compute-bound algorithms require a large number of floating point operations to be performed per byte loaded from memory. After each chunk of data is loaded, the processor remains busy for a long while and the memory system must wait for the processor to finish before transferring the next group of data. As a result, the speed of the processor itself is the rate limiting factor for the performance of the algorithm. Matrix-matrix operations such as those found in Level 3 of the Basic Linear Algebra Subprograms (BLAS) library are generally compute bound. On the other hand, the situation is reversed in memory-bound algorithms. The processor performs only a limited number of

operations per byte, so the processor largely remains idle waiting for data transfers to complete, and the performance of the algorithm is now determined by how fast the data can be transferred from memory. Vector-vector operations such as those in Level 1 BLAS, and matrix-vector operations such as those in Level 2 BLAS are both memory bound on most architectures.

Our current GPU implementation of Anderson acceleration is based on the GPU-optimized BLAS library CuBLAS from Nvidia. The form of the algorithm still follows the structure listed in Algorithms 1 through 3, and the main loop still runs on the CPU. However, except for the small R matrix, the data for the algorithm resides in the GPU RAM, and each vector operation is performed by calling Level 1 BLAS operations on the GPU. For example, each dot product in 2 is done using the CuBLAS function `cublasDdot`. Because the implementation is based on vector-vector operations, the algorithm is expected to be highly memory bound. Therefore, the better flops rate of the GPU is not expected to be helpful, but the much higher memory bandwidth should still give the GPU implementation a performance advantage over a CPU implementation if the bandwidth is well utilized. Unfortunately, invoking an operation on the GPU has a high overhead and each call to a BLAS operation on the GPU incurs about 10 microseconds of latency. Furthermore, the CPU blocks during the execution of each BLAS call (host pointer mode was set) and only the default CUDA stream was used. Therefore, there is no overlap of work between the CPU and GPU, nor between GPU kernels, to hide the overhead of the BLAS calls. To make the cost of each call worthwhile, that overhead must be amortized over a sufficiently large amount of work, i.e. over a sufficiently large vector. We can expect the GPU implementation to perform poorly for small vector sizes, but to perform well if the vector length is sufficiently large to amortize the overhead and allow the high bandwidth to be exploited over many vector elements.

For comparison, a single-node CPU implementation was also developed that keeps the data within the CPU RAM and uses standard BLAS instead of CuBLAS. The bandwidth of the CPU RAM is lower than that of the GPU RAM, but the caching system is superior and the overhead to invoke a BLAS call is comparatively negligible. As such, we can expect the CPU version to outperform the GPU version for small vector lengths, as the cache will be able to hold most of the data and the overhead per BLAS call will dominate the GPU implementation. However, for sufficiently large vector lengths, the data will no longer fit within CPU cache and the overhead per BLAS call will be well amortized on the GPU. The higher bandwidth of the GPU should then allow for higher performance over the CPU version.

The CPU implementation was linked with Intel's optimized BLAS routines provided in the Intel Math Kernel Library (MKL). The code was tested against both single-threaded and multi-threaded versions of the library. Multi-threading allows a greater number of operations to access the memory system at the same time, providing better memory bandwidth utilization at the cost of some thread synchronization overhead.

The implementations were tested on two sets of machines. The primary is representative of a node on a current HPC-grade machine and was used for the performance timings. The secondary machine has a consumer-grade GPU and CPU. It was

not used to gather the primary results, but rather to supplement our understanding of the performance through low-level profiling. Our access rights to the machine allowed use of hardware performance counters that could measure bandwidth usage, which was not possible on the main machine. The configuration of both machines is specified in Table 4.

	Primary	Secondary
CPU:	Intel Xeon E5-2670	Intel i5-3570K
Cores/socket	8	4
# sockets	2	1
Clock rate	2.6 GHz	3.4 GHz
L1 Cache	32 KB	32 KB
L2 Cache	256 KB	256 KB
L3 Cache	20 MB	6 MB
RAM	256 GB DDR3	16 GB DDR3
Memory bandwidth	25.6 GB/s	21.0 GB/s
GPU:	Tesla K40m	GeForce GTX 680
Architecture	Kepler	Kepler
Clock rate	745 MHz	1.18 GHz
L2 Cache	1.5 MB	512 KB
RAM	12 GB GDDR5	4 GB GDDR5
Memory bandwidth	288 GB/s	192 GB/s

Table 4 Configuration of the two machines on which the GPU implementation was tested.

The implementations were tested on the primary machine with four sets of experiments. The first set was run for four Anderson iterations with a window size of $m = 4$ (i.e without QRDelete) for vector lengths ranging from one to ten million, increasing in factors of ten. The remaining experiments were run with sixteen Anderson iterations over the same range of vector lengths, but with window sizes of $m = 4$, $m = 8$, and $m = 16$. Note that in the last case, QRDelete is also not used. Besides running on the GPU, all four sets were run on the CPU using both one thread and sixteen threads. Other numbers of threads were tested but gave results intermediate to the one and sixteen thread cases. As with previous experiments, a dummy function that returned random values was used and the cost of the function was not included in the timings. The results are shown in Figure 2.

The outcome in all four cases is very similar. For vector lengths below ten thousand, the CPU versions require significantly less time than the GPU version in all four experiments. The cost of the CPU implementation remains approximately constant until the vector size reaches a hundred elements, after which the cost begins to converge to a linear increase in cost with vector length. For the GPU version, the cost remains constant until about ten thousand elements per vector due to the high overhead of invoking BLAS calls. Note that in each of the four experiments, the number of vector operations is constant and independent of the vector length. That is why the GPU cost remains constant until the vector length is increased enough for the amount of work per vector to dominate over the overhead per vector operation. Beyond ten thousand elements per vector, the cost slowly approaches a linear

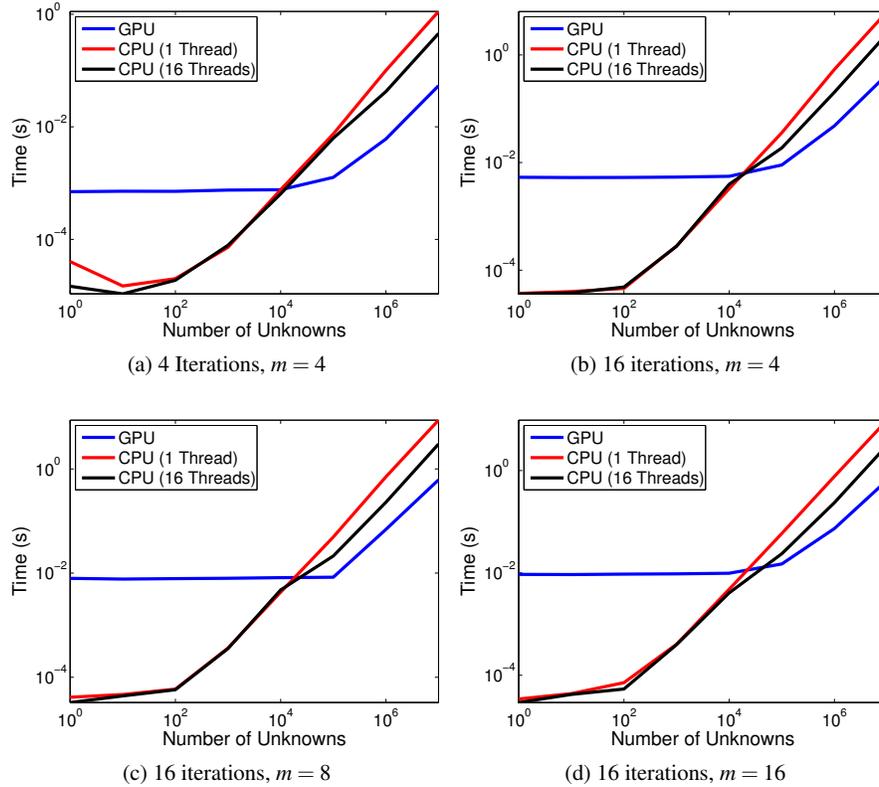


Fig. 2 Performance of a GPU implementation versus a CPU implementation using one and four threads on the primary GPU machine. For sufficiently large vector lengths, the GPU version outperforms the CPU version due to the higher memory bandwidth on the GPU. For smaller vector lengths, the high overhead of invoking BLAS routines prevents the GPU implementation from being competitive.

increase in cost with length. When the vectors are large enough that both the CPU and GPU costs have linear scaling, we expect the ratio in performance to be roughly equal to the ratio in effective memory bandwidth. For the case of four iterations with $m = 4$, the run time for the GPU at ten million unknowns per vector is 5.4×10^{-2} seconds, while it is 4.5×10^{-1} seconds for the multi-threaded CPU case, giving a ratio of about 8.5. For the case of sixteen iterations with a window size of $m = 16$, for a vector length of ten million the GPU run time was 6.5×10^{-1} seconds, while it was 3.1 seconds in the multi-threaded CPU case, resulting in a ratio of only 4.8, implying the GPU bandwidth is not fully utilized. For the sixteen iteration cases with $m = 4$ and $m = 8$, the performance ratios were 5.7 times and 4.9 times respectively compared to the multi-threaded times.

To verify that the higher bandwidth of the GPU is the primary cause for its performance advantage, the CPU implementation was profiled on the 4-core machine using the Intel VTune profiler, and the GPU implementation was profiled on the corresponding GeForce GTX 680 using the Nvidia Visual Profiler. Both profilers are able to measure the memory bandwidth usage directly using low-level hardware performance counters. The qualitative results of the experiments on the second machine were similar to those of Figure 2, except that the difference between the single and multi-threaded CPU cases was much less. For the largest vector size, when computing four iterations with $m = 4$, the difference in run time between the GPU and multi-threaded CPU case was a ratio of 10.0, equal to the peak bandwidth ratio for the machine. For the sixteen iteration cases, the ratios were 8.2 for $m = 4$, 7.8 for $m = 8$, and 7.6 for $m = 16$. Measuring these cases in VTune, for both CPU versions the bandwidth usage over time had the profile of extended periods of high bandwidth during the BLAS calls interspersed with shorter periods of low bandwidth usage between the calls. The peak bandwidth in the multi-threaded case reached over 20 GB/s, which is very close to the peak bandwidth of the machine, while the average bandwidth was 16.1 GB/s. In the single-threaded case, the peak bandwidth reached about 19.5 GB/s, but the profile was considerably less uniform. The average bandwidth was 15.4 GB/s. The lower and less consistent bandwidth was due to only having a single thread access the memory bus, preventing the bandwidth from being consistently held high. Despite the lower bandwidth in the single-threaded case, the run time was always nearly identical to the multi-threaded case on the 4-core machine. This is due to the OpenMP synchronization overhead in the multi-threaded case, resulting in the performance balancing to about the same overall cost. We assume that for the 16-core machine, the overall bandwidth utilization using multiple threads was even better than for the 4-core case, giving a significant net win over a single thread. For the GPU with the largest size of vector, the bandwidth usage within the BLAS calls was generally about 155 GB/s and between the calls the bandwidth was near zero. Almost all of the time was spent within the BLAS calls instead of between. For shorter vector lengths, the percentage of time spent between calls increased, reflecting less amortization of the overhead of invoking the routines, and the bandwidth utilization also fell within the BLAS calls due to the lower amount of concurrency utilizing the memory system. For example, when the vector length was ten thousand, the bandwidth within the BLAS calls fell to only several hundred MB/s.

We conclude that GPUs can provide a significant performance increase over CPU implementations as long as the number of unknowns in the problem is sufficiently high to allow the bandwidth to be exploited. Improvements to the performance of the implementation could come in two forms. The first is to achieve better memory bandwidth utilization. The current implementation has good memory efficiency when the vector lengths are high, but there is some room for improvement. Perhaps a more fruitful approach would be to design an implementation that is less memory bound. Transfers of data are a form of communication and communication avoiding algorithms were in fact first designed to minimize memory cost. Attempting to trade rate of convergence for a reduction in communication cost was not effective in the

MPI case because the amount of inter-node communication was too low to make doing so worthwhile. However, in the on-node case, for large enough vector sizes, almost all of the cost is memory communication. It might be that a GPU implementation of communication avoiding QR factorization or some other communication minimizing approach would give a net performance advantage. We will explore this possibility in future work.

7 Conclusions and Future Work

In this paper we considered whether communication-avoiding QR algorithms in AA could increase efficiency on distributed-memory machines. We found that on 1,000 processors, communication was not significant enough to require communication avoidance. In future work, we will test whether communication becomes more significant when utilizing more processors.

The Anderson iteration can be restarted in a manner similar to GMRES, which mitigates the quadratic growth in cost and memory from an increasing set of past iterates. However, AA can also do an in-place update of the QR factorization to achieve a similar benefit. The latter approach limits the rate of convergence less than the former, but has higher overhead. We tested on only a single problem, but the results suggest that the overhead from the in-place update is high enough that restarting can be modestly beneficial in some cases. We will test on a larger set of problems to see how the balance varies between problems.

Implementation of AA for GPUs can give a sizable performance increase over CPU implementations when the number of unknowns is sufficiently large due to the higher memory bandwidth of GPU memory. We did not find a benefit from MPI-level communication avoidance, but the highly memory bound nature of our current GPU implementation suggests communication avoidance may be useful at the GPU level. We will investigate this in future work.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-PROC-675918.

References

1. SUNDIALS (SUite of Nonlinear and Differential/ALgebraic Solvers). <http://www.llnl.gov/casc/sundials>

2. Anderson, D.G.: Iterative procedures for nonlinear integral equations. *J. Assoc. Comput. Machinery* **12**, 547–560 (1965)
3. Blackford, L.S., Choi, J., Cleary, A., D’Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., et al.: ScaLAPACK users’ guide, vol. 4. *siam* (1997)
4. Brown, P.N., Saad, Y.: Hybrid Krylov methods for nonlinear systems of equations. *SIAM J. Sci. Statist. Comput.* **11**, 450–481 (1990)
5. Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK: A portable linear algebra library for distributed memory computer design issues and performance. In: *Applied Parallel Computing Computations in Physics, Chemistry and Engineering Science*, pp. 95–106. Springer (1996)
6. Demmel, J., Grigori, L., Hoemmen, M., Langou, J.: Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Sci. Comput.* **34**(1), 206–239 (2012). DOI 10.1137/080731992. URL <http://dx.doi.org/10.1137/080731992>
7. Fang, H., Saad, Y.: Two classes of multisection methods for nonlinear acceleration. *Numer. Linear Algebra Appl.* **16**, 197–221 (2009)
8. Hammarling, S., Lucas, C.: Updating the QR factorization and the least squares problem. Tech. rep., The University of Manchester (2008). URL <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.142.2571>
9. Hindmarsh, A.C., Brown, P.N., Grant, K.E., Lee, S.L., Serban, R., Shumaker, D.E., Woodward, C.S.: SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. Math. Softw.* **31**(3), 363–396 (2005). DOI 10.1145/1089014.1089020. URL <http://doi.acm.org/10.1145/1089014.1089020>
10. Jones, J.E., Woodward, C.S.: Preconditioning Newton-Krylov methods for variably saturated flow. In: L.R. Bentley, J.F. Sykes, C. Brebbia, W. Gray, G.F. Pinder (eds.) *Computational Methods in Water Resources*, vol. 1, pp. 101–106. Balkema, Rotterdam (2000)
11. Kelley, C.: *Iterative Methods for Linear and Nonlinear Equations*, *Frontiers in Applied Mathematics*, vol. 16. SIAM, Philadelphia (1995)
12. Knoll, D.A., Keyes, D.E.: Jacobian-free Newton–Krylov methods: a survey of approaches and applications. *J. Comp. Phys.* **193**, 357–397 (2004)
13. Saad, Y., Schultz, M.H.: GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.* **7**(3), 856–869 (1986)
14. Solomonik, E., Ballard, G., Knight, N., Jacquelin, M., Koanantakool, P., Georganas, E., Matthews, D.: NuLAB. <https://github.com/solomonik/NuLAB/>
15. Walker, H.: Anderson acceleration: Algorithms and implementations. Tech. Rep. MS-9-21-45, Worcester Polytechnic Institute (2011)
16. Walker, H.F., Ni, P.: Anderson acceleration for fixed-point iterations. *SIAM J. Numer. Anal.* **49**(4), 1715–1735 (2011). DOI 10.1137/10078356X. URL <http://dx.doi.org/10.1137/10078356X>