Towards a Parallel Algebraic Multigrid Solver Using Partitioned Global Address Space

Niclas Jansson

School of Computer Science and Communication
KTH Royal Institute of Technology
SE-100 44 Stockholm, Sweden
njansson@kth.se

Abstract. The Algebraic Multigrid (AMG) method has over the years developed into an efficient tool for solving unstructured linear systems. The need to solve large industrial problems discretized on unstructured meshes, has been a key motivation for devising a parallel AMG method. Despite some success, the key part of the AMG algorithm; the coarsening step, is far from trivial to parallelize efficiently. We here introduce a novel parallelization of the Ruge-Stüben coarsening algorithm, that retains the good interpolation properties of the original method. Our parallelization is based on the Partitioned Global Address Space (PGAS) abstraction, which allows for a simple, yet efficient implementation. The solver is described in detail and a performance study on a Cray XE6 is presented.

Keywords: Algebraic Multigrid, PGAS, UPC

1 Introduction

Multilevel methods are efficient algorithms for solving large linear systems. For large unstructured problems where geometric based approaches have difficulties the Algebraic multigrid (AMG) method [1] has over the years developed into an efficient tool. While most of the AMG framework can be parallelized in a straightforward way, the important smoothing and coarsening steps are far from trivial to parallelize. Since convergence rate is tightly coupled with the smoother’s performance and the quality of the coarsening, it is important that these steps are carried out correctly.

The problem with parallelizing the traditional coarsening algorithm [1] is its inherently sequential nature. Formulated as a decision problem, which needs to be solved from a global perspective. Implemented using traditional message passing, solving this problem becomes difficult. The key issue with a message passing based parallelization is the lack of a global knowledge between the processing elements (PEs). For example, if a PE solves part of the problem, since the dependencies are global, this change has to be communicated to all neighboring PEs in order to ensure correctness. Since communication is expensive such a solution would not be scalable, hence several trade-offs have to be made.
Due to the challenges of exascale computing, there has been a push forward on developing new programming models that can handle large amount of fine grained parallelism. One good example is the Partitioned Global Address Space (PGAS) languages. Based on the abstraction of a global shared address space, built on top of the distributed global memory it is a simple and elegant model, especially for algorithms with challenging data dependencies.

In this paper we address the issues of message passing based coarsening and smoothing algorithms by implementing a novel parallelization using the PGAS programming model. This work represent our first step in the development of a PGAS based AMG solver. The outline of the paper is the following: In §2 a short overview of AMG is given. In §3 a brief introduction to AMG coarsening is given and related work on parallelization is discussed. In §4 we present our novel approach to parallel coarsening and implementation details and in §5 we discuss our AMG solver and parallelization of the smoother. A performance evaluation is given in §6, and we give conclusions and outline future work in §7.

2 Algebraic Multigrid

In order to better understand the coarsening process, we begin by outlining the basic AMG algorithm. For a problem,

\[ Ax = b \]  

where \( A \) is an \( n \times n \) matrix with entries \( a_{ij} \) and \( x, b \) are vectors of size \( n \). The idea behind multilevel methods such as AMG is to eliminate “smooth errors” by solving the residual equation \( Ae = r \) on a coarser representation of the problem, and interpolate it back to correct the solution of the real problem (1) by \( x := x + e \). By utilizing several layers of coarser representations of (1) and cycling down and back up between them results in a very efficient method.

Before any AMG solver can begin a series of operators \( A^k \), interpolation operators \( I_{k+1}^k \) and restriction operators \( I_k^{k+1} \) need to be constructed, where we let \( \Omega^k \) be the set of components of \( x \) at level \( k \). In order to define these operators, the AMG method always begins with a setup phase.

**Algorithm 1 AMG Setup phase:**

Let \( k = 1 \)

while \(|\Omega^k| > \text{stopping size}\) do

- Partition \( \Omega^k \) into disjoint sets \( C^k \) and \( F^k \)
- Set \( \Omega^{k+1} = C^k \)
- Define interpolation \( I_{k+1}^k \) and restriction \( I_k^{k+1} = (I_{k+1}^k)^T \)
- Construct next level \( A^{k+1} = I_k^{k+1}A^k I_{k+1}^k \), and set \( k = k + 1 \)

end while

For the AMG method to work efficiently, a set of rules has to be followed when \( \Omega^k \) is partitioned. This step is often referred to as the coarsening step. Once these operators have been defined, the solution step can be executed as a
recursively defined cycle. The cycle is repeated until the error is below a certain tolerance. Different cycling algorithms can be used, with various complexity. In Algorithm 2 the basic V-cycle is described. For a more detailed description of more complex cycling algorithms, we refer to [2] and [1].

Algorithm 2 Multigrid V-cycle:

\[
\text{if } k = \text{coarsest level then}
\quad \text{Solve } A^k x^k = f^k
\]
\[
\text{else}
\quad \text{Relax } \mu_1 \text{ times on } A^k x^k = f^k
\quad \text{Set } x^{k+1} = 0, f^{k+1} = I_{k+1}^k (f^k - A^k x^k)
\quad \text{Apply V-cycle on level } k + 1
\quad \text{Correct the solution by } x^k = x^k + I_{k+1}^k x^{k+1}
\quad \text{Relax } \mu_2 \text{ times on } A^k x^k = f^k
\text{end if}
\]

3 AMG coarsening

When deriving a coarsening method the question is how to partition the problem such that the matrix coefficients in \( A^{k+1} \) are the most important ones for eliminating certain errors at the current level \( k \). Since the error varies differently depending on how strongly the unknowns are coupled to each other, we need a measure to distinguish which points should be part of a coarser level \( k + 1 \). One measure we can use is to say that an unknown \( i \) strongly depends on \( j \) if,

\[
-a_{ij} \geq \theta \max \{ -a_{ik} \}
\]

given a threshold \( 0 < \theta \leq 1 \). Using the strength measure (2) the components of \( x \) (points) are partitioned into two disjoint sets, \( \Omega = C \cup F \), where \( C \) is the coarse set of unknowns and \( F \) is the set which will be interpolated from \( C \). The classic coarsening algorithm Ruge-Stüben (RS) [1] derives a partitioning by adhering to two criteria for choosing \( C \) and \( F \):

\begin{enumerate}
  \item [C1:] For each point \( j \) that strongly influences an \( F \)-point \( i \), \( j \) is either a \( C \)-point or it strongly depends on a \( C \)-point \( l \) that also strongly influences \( i \).
  \item [C2:] \( C \) should be a maximal subset of all points with the property that no two \( C \) points are strongly connected to each other.
\end{enumerate}

C1 relates to the quality of the interpolation between levels and C2 limits the size of the coarser level.

The algorithm consists of two phases, which are easily described using a graph. If we represent the matrix as a graph, where each node corresponds to a matrix coefficient. The first preliminary \( C \)-point selection assigns for each point \( i \) a measure \( \lambda_i \), the number of neighbors that are strongly influenced by \( i \). The
point with the largest $\lambda_i$ is then selected as the first $C$–point and all points that strongly depend on $i$ are selected as $F$–points, and all neighbors that are strongly dependent on the new $F$–points have their measure increased by one. This procedure is then repeated, with all unassigned points until all points have been selected as either $C$ or $F$ points.

After the first phase $\Omega$ has already been partitioned into two disjoint sets $C$ and $F$. However, the algorithm does not guarantee criterion C1, and might have several $F$–$F$ connections which violate the criteria. Therefore, the second phase of the algorithm sequentially tests each $F$–point to make sure that C1 is enforced. If a $F$–point is found that violates the criteria, the point is made into an $C$–point. After all $F$–points have been tested, the partitioning of $\Omega$ adheres to C1 but has not enforced condition C2.

### 3.1 Parallel coarsening

Parallelizing RS coarsening is far from easy, mostly due to the sequential nature of the algorithm. The preliminary $C$–point selection is the easiest one, here the problem is to assign a correct measure $\lambda_i$ to each point. In parallel the matrix’s coefficients are distributed in some way, and neighbors of a point $i$ might not reside on the same PE as $i$. For a message passing based implementation, the problem is solved by defining a halo region with copies of the shared coefficients.

The challenging part is to parallelize the second phase without violating C1. Recall from the description of the algorithm, that during the second phase each $F$–point is checked against the criteria, and if it violates it, the point is made into an $C$–point, and here lies the troublesome part of the parallelization. Assuming the coarsening is running in parallel on a distributed memory machine, and the code has been implemented using message passing. If a shared (between PE) $F$–point is made into a $C$–point all neighboring PE’s must be notified of the change, otherwise $F$ points may be unnecessarily converted into $C$ points. This notification must then be made in the form of a message, which has to be received by some or in the worst case all the other PEs. Since this has to be done for all shared points during the second phase it leads to complicated and highly unstructured communication patterns that will affect performance.

Various different approaches have been taken to parallelize RS on distributed memory machines [3]. The simplest one executes the first and second pass of the algorithm on the part that the PE owns, without any special treatment for the shared points, which is likely to violate C1 between PEs.

In order to enforce the criteria between PEs, the original RS algorithm is extended with a third pass on the shared points in order to ensure C1, this leads to the so called RS3 algorithm [3]. This fix up phase is actually the second phase again but only applied on the shared points. However, the third phase may still leave some $F$–$F$ connections between PE (since it is still a local procedure) which have to be resolved. This can be done either by voting, or based on the PE’s id or the extreme case, make all violating $F$ points into $C$ points. In either case, the underlying issue is the programming model, which lacks a global view of the distributed data.
The difficulty with deriving an efficient RS coarsening algorithm in parallel has lead to the development of several more easily parallelized algorithms. Popular methods in these categories are for example the Parallel Modified Independent Set (PMIS) [4] and the Hybrid Modified Independent Set (HMIS) [4]. These algorithms are inherently parallel, where each PE builds an independent set for the unknown components it owns. The PMIS algorithm coarsens only using the set based algorithm while HMIS combines the first pass of RS (for the local unknowns) with PMIS to cleanup the shared boundary. The parallelism in these methods comes with a price, namely a violation of criteria C1. Instead both methods enforce a less stringent requirement [4]:

\[ C1': \text{Each } F\text{–point needs to strongly depend on at least one } C\text{–point.} \]

The penalty of not enforcing C1 comes in the form of vanishing terms in the interpolation formula. Therefore, for PMIS and HMIS coarsening schemes the interpolation formula needs to be modified in order to handle these cases, which in turn could lead to poor convergence rates.

4 Global address space based Ruge-Stüben coarsening

Our approach to parallelize Ruge-Stüben coarsening is based on the PGAS programming model. Based on the abstraction of a global address space built on top of the distributed global memory, any PE may directly read or write at any remote memory location without explicitly communicating with any other PE.

If the \( C \) and \( F \) points are placed in global memory, and the second phase of the algorithm is executed, the switch from an \( F \)–point to a \( C \)–point is immediately visible on all PEs, without any explicit communication. Thus, the second phase of the algorithm will produce a coarsening which adheres to C1 without the need of any complicated third pass.

4.1 Implementation

For the PGAS based implementation we use Unified Parallel C (UPC) [5], a C like language that extends ISO C99 with PGAS constructs. The solver is implemented in the sparse matrix library JANPACK [6][7]. The sparse matrices are based on a row-wise distribution, with the matrix entries placed in global memory for easy access by any thread. The AMG specific data structures are mostly based on red-black trees, such as the set of \( C \) and \( F \) points. For the measure \( \lambda_i \) we use a red-black forest, where each node with key \( q \) (in the tree) contains a red-black tree of the matrix coefficients with the measure \( \lambda_i = q \). With this unusual representation we reduce the search time to find the maximum \( \lambda_i \) in the first phase. Note that these trees are placed in local memory, and are thus not accessible by the other threads. In order to enable the parallel coarsening as proposed above, each PE maintains a list of \( C \) and \( F \) variables in global memory. These lists are also protected by a strict access policy such that false sharing or race conditions are prevented. Let \( Cg \) be a list in the global memory, which is
set to \( C_g(l) = 1 \) if point \( l \in C \) and zero otherwise. The coarsening algorithm can then be expressed as:

**Algorithm 3 Parallel Ruge-Stüberen:**

Let \( U = \Omega^k \) and \( C_g(\cdot) = 0 \)

/* First phase */

\[ \text{while } U \neq \emptyset \text{ do} \]
- Pick an \( i \in U \) with maximal \( \lambda_i \)
- Set \( U = U - \{i\} \), \( C = C + \{i\} \) and \( C_g(i) = 1 \)
- Add all points \( j \) which strongly depends on \( i \) to \( F \)
- Increase the measure \( \lambda_l \) for all points \( l \) that are strongly dependent on \( j \)
- Decrease the measure \( \lambda_m \) for all points \( m \) that are strongly dependent on \( i \)

end while

Barrier

/* Second phase */

\[ \text{for all } i \in F \text{ do} \]
- if \( i \) violates criteria \( C1 \) then
  - Memory fence
  - \( F = F - \{i\} \), \( C = C + \{i\} \) and \( C_g(i) = 1 \)
  - Memory fence

end if

end for

As we can see in Algorithm 3, there are few regions with explicit parallel code. The first phase, can be executed without any synchronization, except at the end, in order to ensure that \( C_g \)’s state is synchronized across all threads. In the second phase, while checking if criteria \( C1 \) holds, a thread may need to consult a remote location in \( C_g \) in order to determine if a non local point is a \( C \) or an \( F \) point. Furthermore, \( C_g \) might also be updated during this phase if an \( F \) point is found that violates \( C1 \). Therefore, as mention above we use a strict access policy (memory fences) in order to avoid any race conditions on \( C_g \).

Interpolation weights are constructed from the matrix coefficients of the nearest neighbors. Since the matrix is already in the global memory this step can be performed in a straightforward way. For constructing the interpolation operator each \( C \) point has to be given a unique number (corresponding to the index in the new operator \( A^{k+1} \)). We compute a prefix sum of \( |C| \) to find the starting offset for each thread, which then labels its \( C \) points linearly from the given offset. The new indices are stored in a hash table, together with indices for any remote \( C \) point (fetched from a list in the global memory) needed to interpolate the thread’s \( F \) points. The matrix–matrix multiplication required to construct the operator for a coarser level is implemented as a PGAS version of Gustavson’s sparse matrix multiplication algorithm [8].

### 4.2 Load balancing

Since the operator \( A^{k+1} \) will be smaller and smaller for each coarsening level \( k \), it will at a point become too small to be represented using a certain amount of PEs.
To overcome this problem, the coarsening process implements a load balancing routine, that redistributes the new operator on a subset of the PEs using a load balanced linear distribution [9]. The load balancing routine is triggered when the average amount of rows per PE is below a certain threshold, a new distribution is then computed and a subset of the PEs becomes responsible for holding the rows for $A^{k+1}$, effectively, moving the operator towards a single PE for the coarser levels.

Implementing such a load balancing scheme using UPC turned out to be quite challenging. Using MPI, the operator can be defined using a communicator, a subset of all the MPI ranks. UPC however, does not incorporate any concept similar of communicators, and the user is forced to use either all or none of the UPC threads, especially troublesome when using the builtin collective operations. Our implementation therefore assigns zero rows to PEs which are not part of the new operator. Hence, the operator are always defined on all PEs but only a subset of the PEs is assigned a set of rows. In order to reduce the amount of communication during the required collective operations (prefix sums etc.), we implemented an own version of the collectives (operating only on a subset of UPC threads) that we needed and switch from the one provided by UPC to our own as soon as we pass the load balancing threshold.

5 Multigrid cycle

Parallelizing most of the parts in the V–cycle described in Algorithm 2 is fairly straightforward, most of the operations are matrix–vector products which already existed in the linear algebra library. However, since the operator $A^{k+1}$ might be distributed across a different number of PEs than $A^k$, redistribution routines had to be implemented for vectors, such that the residual computation and correction of the solution could be computed.

The most challenging part of the cycling algorithm is to parallelize the smoother, used to relax errors in the down- and upward pass of the cycle. Traditionally, Gauss–Seidel is the smoother of choice for serial multigrid solvers since it has good smoothing properties given it’s computational cost, it can be formulated as:

$$x_i^k = \left( b_i - \sum_{j<i} a_{ij}x_j^k - \sum_{j>i} a_{ij}x_j^{k-1} \right) / a_{ii}$$

(3)

where $k$ is the iteration, $i$ and $j$ are components of $x, b$ and $a$.

One of the main problems in devising an efficient parallel Gauss–Seidel method is the dependency on the previously computed values $x_j^k$ and of course, off PE’s dependencies. Often a Processor Block or also referred to as hybrid Gauss–Seidel [10][3][11] is employed, whereas the relaxation is performed on the points each PE owns, and the shared points are updated only after each relaxation sweep, in a sense similar to a block Jacobi method. Often the unknowns are visited in a certain order to reduce data dependencies ($x_j^k$ in (3)) and to improve the relaxation, for example first all the $C$ points and then $F$ points on the
down cycle, and first \( F \) points and then \( C \) points on the up cycle, often referred to as a hybrid \( CF \)-Gauss-Seidel [3]. However, since the hybrid smoother only considers the PE’s local variables, multigrid convergence rates might suffer [11], a trade-off due to the programming model used.

With the entire operator \( A^k \) placed in global memory, our approach for devising a parallel smoother was straightforward. Based on the basic formulation (3), we implemented \( CF \)-Gauss-Seidel, which operated on all the unknown. If the relaxation needed an unknown from another PE, the value is loaded from the global memory without any explicit communication. Furthermore, since updated variables \( (x^k_i \text{ in (3)}) \) already are stored in the global memory, the PGAS based \( CF \)-Gauss-Seidel has similar convergence properties as a serial implementation. However, the price for the improved relaxation is a large amount of fine grained parallelism, while fetching off PE unknowns. But this cost should be amortized by improved multigrid convergence in parallel.

6 Performance evaluation

The parallel AMG solver was implemented and evaluated on the 1516 node Cray XE6, called Lindgren, located at PDC/KTH. Each node consists of two 12-core AMD “Magny-Cours” running at 2.1GHz, equipped with 32GB of RAM. The Cray XE6 is especially well suited for our work since its Gemini interconnect provides hardware accelerated PGAS support, and it has also a mature toolchain for PGAS development. In this work we used the Cray Compiler Environment (CCE) version 8.0.6. On this machine we have also performed a set of numerical experiments in order to evaluate the scalability and quality of our implementation.

6.1 Benchmark problem

As a benchmark problem for all evaluations we solved Poisson’s equation on an unstructured mesh of the unit square with a dolphin shaped hole, applying homogeneous Dirichlet boundary conditions on one side and different Neumann boundary conditions on the remaining sides,

\[
\begin{align*}
-\Delta u(x, y) &= f(x, y), & (x, y) &\in \Omega, \\
 u(x, y) &= 0, & (x, y) &\in \Gamma_0, \\
 \partial_n u(x, y) &= g(x, y), & (x, y) &\in \Gamma_1, \\
 \partial_n u(x, y) &= 0, & (x, y) &\in \partial\Omega \setminus (\Gamma_0 \cup \Gamma_1),
\end{align*}
\]

(4)

where, \( f(x, y) = 500 \exp(-(x-0.5)^2+(y-0.5)^2)/0.02 \) and \( g(x, y) = 25 \sin(5\pi y) \). The PDE was discretized using the HPC branch [12] of the finite element software DOLFIN [13] running in hybrid MPI+PGAS mode. The assembled finite element stiffness matrix was then fed into the AMG solver or directly to the AMG coarsening algorithm, depending on the experiment.
6.2 Coarsening quality

Before we conduct a scalability study of our new coarsening algorithm we wish to demonstrate its coarsening quality. When applying AMG coarsening in parallel it is not always the case that the resulting coarsening is independent of the number of PEs employed. Depending on the algorithm used, there can be various artifacts from the boundaries between PEs. Since our algorithm partitions $\Omega$ using a global view of the problem, these artifacts vanish. In Figure 1(a) we show the coarsening produced after RS has been applied in serial to the two dimensional Laplace equation on an unstructured mesh. The parallel coarsening in Figure 1(b) is of course different (the algorithm is applied in a different order, concurrently in all mesh partitions) but there are no artifacts from the shared boundary, as it can be seen when comparing the parallel results with the data decomposition for the problem in Figure 1(c).

6.3 Coarsening scalability

To measure the scalability of the new Ruge-Stüben parallelization, we assembled the finite element stiffness matrix for the corresponding benchmark problem (4) in DOLFIN. The assembled finite element stiffness matrix was then fed into the AMG coarsening algorithm. The total time to compute one level was measured, in practice the time it took to execute Algorithm 1. The experiments were executed on $2^i$ nodes (24 PEs per node), for $i = 1, 2, \ldots, 5$ (48, 96, $\ldots$, 768 PEs) using a mesh with roughly 2.8M vertices.

To measure the cost of issuing several remote memory accesses during the second pass of Ruge-Stüben, we also implemented and performed the experiments for PMIS and HMIS coarsening, which only uses remote memory for computing...
In Figure 2 and Table 1 we present the execution time for one level of coarsening, using the three different algorithms. The results shows that both PMIS and HMIS coarsening performs well, an expected result since the algorithms are inherently parallel with few synchronization points. More interesting is the result for the Ruge-Stüben algorithm which performs as well as both set based algorithms. This result indicates that the remote memory accesses in Ruge-Stüben were less expensive than expected, and did not have any negative impact on the performance. Furthermore, in Table 1 we also present the grid complexity, defined as:

\[ C_g := \sum_l \frac{n_l}{n_1}, \]  

(5)

where \( n_l \) denote the number of unknowns on level \( l \). In our case the grid complexity is an interesting measure to see how much the coarsening differs when we increase the number of PEs. As observed in Table 1 \( C_g \) stays more or less
constant as we increase the number of PEs, which indicates that the coarsening, is as we have observed for the smaller problem, independent of the data distribution.

To put the result into perspective, set based coarsening algorithms such as PMIS and HMIS are often employed due to their low (parallel) coarsening cost. But due to the less efficient interpolation, one usually has to perform more multigrid iterations to obtain a solution. However, with a low cost Ruge-Stüben coarsening, that also has better interpolation properties the overall cost of solving an equation should be lower.

6.4 AMG Scalability

The last experiment we performed was a weak scalability study for solving Poisson’s equation (4) using different problem sizes and number of PEs. We ran the experiment for six different configurations, each using the same unit square as in previous experiments, but with various amount of uniform mesh refinement applied.

For each configuration, we used DOLFIN in hybrid MPI+PGAS mode, compiled to use JANPACK as its linear algebra backend. Inside the AMG solver, the Ruge-Stüben coarsening was allowed to create a maximum of 20 levels, with a minimum operator threshold of 50 unknowns. On each level a threshold value of $\theta = 0.15$ was used for the strength measure (2). We used a V–cycle within the multigrid solver together with a $CF$–Gauss–Seidel smoother performing five relaxation sweeps ($\mu_1 = \mu_2 = 5$ in Algorithm 2). The coarsest level was solved for using a preconditioned BiCGSTAB Krylov subspace method with a D-ILU preconditioner [14].

For all runs we measured the number of unknowns ($n$), levels ($l$), number of V–cycles (Iters), coarsening time ($t_{\text{setup}}$), time spent in solver performing V–cycles ($t_{\text{solve}}$), total time ($t_{\text{tot}}$), grid complexity ($C_g$) (see (5)) and operator complexity defined as,

$$C_{\text{op}} := \sum_l \frac{n_{z_l}}{n_{z_1}},$$

where $n_{z_l}$ denotes the number of non–zero entries in the operator at level $l$.

As a reference, all configurations were also solved for using a preconditioned Krylov solver, BiCGSTAB + D-ILU, and recorded the amount of Krylov iterations and total time $t_{\text{tot}}$. For both solvers, AMG and Krylov, the same convergence criteria were used.

In Table 2 we present all the results from our experiments. According to theory we know that multigrid methods has a low complexity, and the solver iterations should stay almost constant, independent of the problem size [15],[16]. In Figure 3 (left) where we compared the amount of iterations for the AMG and Krylov solver, we can see that our AMG solver almost achieve the low complexity predicted by theory, regardless of the number of PEs the problem is solved on. We observed that the Krylov solver requires more and more iterations to converge as the problem size grows. However, each multigrid iteration also becomes
more and more expensive as the problem size grows, consistent with the total runtime presented in Figure 3 (right). Furthermore, Table 2 also demonstrates that the grid and operator complexities are constant for all configuration. Since these metrics does not grow, it is a good indicator that our coarsening is fairly independent on the number of PEs used. Also, coarsening time ($t_{\text{setup}}$) was less than the multigrid cycling time ($t_{\text{solve}}$) for all configurations, indicating that none of the components exhibits more negative effect from parallelism than the other.

To determine how much the communication affected our implementation we also measured weak scalability. Since the benchmark problem was unstructured, it is difficult to perform a traditional weak scaling study, where the problem size is doubled when the number of PEs is doubled. Instead we took the results in Table 2, and computed, for each configuration the time to process an unknown $t_n$ as:

$$ t_n = \frac{t_p p}{n}, $$

where $t_p$ is the time to perform an operation on $p$ PEs, given a problem size of $n$ unknowns. Using the timings in Table 2, we computed the time per unknown
Fig. 4. Weak scaling results for the different AMG components, Ruge-Stüben coarsening and V–cycling (left) and for the total runtime (right) when solving the benchmark problem (4) using an AMG and a Krylov solver.

<table>
<thead>
<tr>
<th>PEs</th>
<th>AMG</th>
<th>Krylov</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Coarsening</td>
<td>V–cycle</td>
</tr>
<tr>
<td>1</td>
<td>3.557e-05</td>
<td>1.186e-04</td>
</tr>
<tr>
<td>48</td>
<td>4.560e-04</td>
<td>1.088e-03</td>
</tr>
<tr>
<td>192</td>
<td>1.110e-03</td>
<td>1.872e-03</td>
</tr>
<tr>
<td>384</td>
<td>1.714e-03</td>
<td>4.227e-03</td>
</tr>
<tr>
<td>768</td>
<td>2.023e-03</td>
<td>2.555e-03</td>
</tr>
</tbody>
</table>

Table 3. Weak scaling results for the different AMG components and for the total runtime when solving the benchmark problem (4) using an AMG and a Krylov solver.

In the ideal case, \( t_n \) should be constant as we scale up the problem size. However, since we also add more PEs, and introduce communication costs, this will affect the result. This effect is easy to observe in Figure 4, where \( t_n \) initially makes a large jump since the solver goes from serial to a parallel implementation. But for the last data points we observe that \( t_n \) does not change much, which indicates that the communication cost are well balanced with the computational costs. Our results indicate that this is true for both the AMG components (Figure 4, left) as well as for the entire AMG solver (Figure 4, right). Furthermore, our results for the entire solver Figure 4 (right) indicates that the cost per unknown in the AMG solver tends to be more constant as we increase the problem size, but continues to grow for the Krylov solver.
7 Summary and future work

In this work we have investigated the applicability of partitioned global address space languages for formulating a parallel algebraic multigrid solver. Using the global address space abstraction of PGAS we were able to formulate a novel parallelization of the traditional Ruge-Stüben coarsening algorithm, which retains the good interpolation properties of a serial Ruge-Stüben coarsening. Our performance evaluation shows that the new algorithm performs well, and is competitive with other less costly, set based coarsening algorithms.

The entire multigrid V–cycle has also been implemented, together with a parallel CF–Gauss-Seidel smoother. Compared to other work, the smoother operates on all unknowns without any significant performance degradation. Our numerical experiment results indicate that the PGAS based solver performs well both in runtime and amount of solver iterations for various problem sizes, and results in significant improvements in performance compared to a preconditioned Krylov subspace method.

This work represents our first steps towards a PGAS based AMG solver. Based on the encouraging first results, our current work is focused on improving load balancing in the coarsening algorithms, in particular for the intermediate levels, on which most of the coarsening time is spent today.

To conclude, our results demonstrate that PGAS languages can be used to devise novel parallelization algorithms for problems with large amount of fine grained parallelism. Problems that previously have been considered too complicated or too expensive to solve efficiently using traditional message passing.

Acknowledgments The author would like to acknowledge the financial support from the Swedish Foundation for Strategic Research. The research was performed on resources provided by the Swedish National Infrastructure for Computing (SNIC) at PDC - Center for High-Performance Computing.

References