TeaLeaf: a mini-application to enable design-space explorations for iterative sparse linear solvers

Matt Martineau*, Tom Deakin*, Grzegorz Pawelczak*, Simon McIntosh-Smith*, Wayne Gaudin†, Paul Garrett†, Wei Liu†, Richard Smedley-Stevenson† and David Beckingsale‡

*University of Bristol, Bristol, UK
†AWE, Aldermaston, UK
‡Lawrence Livermore National Laboratory, Livermore, CA, USA

Abstract—**Iterative sparse linear solvers are an important class of algorithm in high performance computing, and form a crucial component of many scientific codes. As intra and inter node parallelism continues to increase rapidly, the design of new, scalable solvers which can target next generation architectures becomes increasingly important. In this work we present TeaLeaf, a recent mini-app constructed to explore design space choices for highly scalable solvers. We then use TeaLeaf to compare the standard CG algorithm with a Chebyshev Polynomially Preconditioned Conjugate Gradient (CPPCG) iterative sparse linear solver. CPPCG is a communication-avoiding algorithm, requiring less global communication than previous approaches. TeaLeaf includes support for many-core processors, such as GPUs and Xeon Phi, and we include strong scaling results across a range of world-leading Petascale supercomputers, including Titan and Piz Daint.**

I. Introduction

HPC architectures are widely expected to continue their adoption of ever more parallel computer architectures in order to deliver greater performance. This increase in parallelism is at every level of the system: CPUs and GPUs are already capable of thousand-way data parallelism, and the largest machines today can include tens of thousands of such highly parallel nodes. This significant and sustained increase in parallelism creates tremendous pressure to find new algorithms which can efficiently exploit next-generation systems. In this paper we present TeaLeaf, a recent mini-app designed to enable the exploration of highly scalable, many-core aware algorithms for iterative sparse linear solvers. We demonstrate TeaLeaf's effectiveness by way of an investigation into strong scaling on two of the world's largest many-core based supercomputers: Titan at Oak Ridge, and Piz Daint at CSCS.

We use a diffusion problem as the basis for our strong scaling study, since the diffusion equation is commonly used in multiple fields of science. Random walk processes occur in diverse areas in nature, with examples found in transport across a biological cell wall [1], thermal conduction [2] and neutrino transport in a supernova [3]. Thus the ability to solve the diffusion equation quickly plays an essential role in a wide range of scientific research.

Iterative sparse linear solvers are the method of choice for solving the heat diffusion problem on distributed memory systems. The move towards the HPCG benchmark[1] as a complementary performance metric to LINPACK[2] highlights the importance of these sparse linear solvers in the high performance computing community. This paper will concentrate on a simple heat conduction system, but the lessons and techniques applied are equally relevant to other random walk systems.

Exchanges of information between a node and its neighbors, the so-called *halo exchange*, has been shown to be an efficient approach for both weak scaling and strong scaling heat diffusion problems [4]. For an explicit method this is all that is required. However a sparse linear solve also requires the calculation of the dot product, which needs data that is non-local to each node's neighborhood; it is this non-local data exchange which stresses the network significantly for this class of sparse solver, often proving to be the limiting factor when scaling over very large systems with thousands of nodes. We therefore need to consider "communication avoiding" sparse solver methods, which, for example, could replace the system-wide dot product operations with halo exchanges. Such communication avoiding techniques are expected to improve scalability, and even if they are less efficient at calculating the solution at low node count, they should ultimately reduce run time for strong scaling problems at large node counts (thousands of nodes or greater).

The current best in class iterative solvers, such as algebraic or geometric multigrid (AMG, GAMG), have concentrated on minimizing iteration count, but require complicated relaxation and prolongation operators that will stress the interconnect significantly and have high set up costs. These solvers tend to perform well at low node counts (tens to hundreds) and have also been shown to weak scale well, but strong scaling is very dependent on the minimization of communication between nodes, and this reduces their parallel efficiency as node counts increase. Our goal for this work is to use TeaLeaf to explore the design of a linear solver that is highly concurrent at the node level, which minimizes communications, and yet is still accurate and robust.

In this work we make the following novel contributions:

1) We introduce TeaLeaf, an open-source mini-app released

[1]http://hpcg-benchmark.org
[2]http://www.netlib.org/linpack

as part of the R&D 100 award-winning Mantevo suite. TeaLeaf is a vehicle to explore the design of new solvers, and for comparing the efficiency of new parallel programming languages. TeaLeaf includes many-core support designed in from the start.
2) We describe a communication-avoiding CPPCG sparse iterative solver within the TeaLeaf framework.
3) We present good strong scaling results for the TeaLeaf CPPCG implementation, using a representative heat diffusion problem running on three world-leading Petascale machines: Titan at Oak Ridge, Piz Daint at CSCS, and Spruce at AWE. The results demonstrate much greater performance and strong scaling for CPPCG versus existing CG-based sparse iterative solvers.

II. Mini-Apps and TeaLeaf

In order to provide an agile research vehicle to investigate our potential design space, we have used mini-apps to evaluate our solvers and techniques. Mini-applications are small, self-contained programs that embody essential performance characteristics of key applications [5].

For this study we developed the *TeaLeaf* mini-app, which has been included as part of Sandia's Mantevo[3] mini-app benchmark suite [5]. TeaLeaf solves the linear heat conduction equation on a spatially decomposed regular grid in two and three dimensions via five and seven point finite difference stencils respectively, using implicit solvers to invert the linear system. For space reasons, this paper focuses on the two dimensional implementation, but the 3D results are similar. In TeaLeaf, temperatures are stored at the cell centers. A conduction coefficient is calculated that is equal to the cell centered density, which is then averaged to each face of the cell for use in the solution. The solve is carried out using an implicit method due to the severe time step limitations imposed by the stability criteria of an explicit solution for a parabolic partial differential equation. The implicit method requires the solution of a system of linear equations which form a regular sparse matrix with a well defined structure.

Normally, because of the complexity of state of the art methods, third party solvers are used to invert the system of linear equations, but few of these solvers are as-yet efficient on many-core hardware. TeaLeaf therefore integrates a range of stand-alone solvers, including Jacobi, Chebyshev and CG. These methods have the advantage of supporting a matrix free approach. The matrix free method gives enhanced performance since data is directly accessed in the original mesh and no explicit matrix construction is required. However, the matrix free approach also makes preconditioning more difficult. We address this issue by introducing a polynomial preconditioned, matrix free, CG (PPCG) method, that uses the Chebyshev solver as the preconditioner, an approach we term CPPCG. This method's main advantage is that it reduces global communication costs, benefiting strong scaling performance.

[3]https://mantevo.org

All of these methods in TeaLeaf have been written in FORTRAN with highly optimized OpenMP and MPI implementations. To investigate the many-core ready capabilities of our approach, we have also ported all of TeaLeaf's methods to OpenCL and CUDA for execution on GPUs. To enable comparisons with existing approaches, TeaLeaf can also invoke third party linear solvers, including PETSc [6], Trilinos [7] and Hypre [8].

III. CPPCG: Communication Avoiding CG

A. The Conjugate Gradient Method

The CG method is well suited to large scale parallel processing, as the only non-local primitives employed by CG are sparse matrix-vector products and dot products. Given discretization schemes that lead to short range stencils, then the sparse matrix-vector product only requires halo data from nearby nodes. The scaling bottleneck is then the MPI reduction operations required to sum over local contributions to the dot products. An optimal implementation of these reductions will ensure that the latency overhead scales logarithmically with the number of nodes. Thus parallel efficiency should fall off only logarithmically given suitable network provisioning. The local operations are vector triads typically requiring two loads and one store per (one or two) floating point operations and as such are local memory bandwidth limited.

B. Polynomial Preconditioning

The idea behind polynomial preconditioning is to premultiply the system equation $A\vec{x} = \vec{b}$ by a polynomial approximation to the inverse of A^{-1}; that is, we will solve:

$$B(A)A\vec{x} = B(A)\vec{b} \tag{1}$$

where $B(\lambda)$ is a pre-conditioning polynomial and $B(A)$ is the associated preconditioner.

As noted by O'Leary [9], among all polynomials of degree at most k, the CG method is optimal, in the sense of minimizing the A-norm of the error. Hence choosing a polynomial preconditioner cannot speed convergence, consequently the number of sparse matrix-vector multiplies cannot decrease. Nevertheless, polynomial preconditioning is useful in reducing the time taken until convergence as the number of CG iterations may be reduced, since many sparse matrix vector operations are undertaken at each step. Quoting directly from O'Leary [9],

1) On a message passing parallel architecture, many problems can be partitioned so that the matrix-vector multiplication requires only local communication among processors, while the accumulation of inner products for the CG parameters requires global communication.
2) On a machine with a memory hierarchy, bringing the matrix A into the highest speed memory to prepare for a matrix-vector product may be a time-consuming operation. Reducing the number of times this is done is an important consideration, and the polynomial preconditioned algorithm is designed to use the matrix for several multiplications at a time.

3) On vector processors, if a matrix-vector multiplication is efficient, then so is forming the product of a matrix polynomial with a vector, and efficiencies in the matrix-vector product are automatically exploited in the preconditioning.

This allows for significant reductions in the overheads associated with the global dot products especially at very high core counts. It also provides further opportunities to exploit the severely limited memory bandwidth available today.

C. Chebyshev Polynomials

Following Ashby, Manteuffel, and Otto [10] we choose to pre-condition the CG method with a shifted and scaled Chebyshev polynomial, $B(\lambda)$, where

$$B(\lambda)\lambda = 1 - \frac{T_m(\xi(\lambda))}{T_m(\xi(0)}. \tag{2}$$

$T_m(x)$ is the mth Chebyshev polynomial of the first kind [11]. The mapping function $\xi(\lambda) : [0, \infty] \to [-1, +1]$ is given by

$$\xi(\lambda) = \frac{2\lambda - (\lambda_{max} + \lambda_{min})}{(\lambda_{max} - \lambda_{min})}, \tag{3}$$

where λ_{min} and λ_{max} are the smallest and largest eigenvalues of the system matrix A. The implementation is based upon the application of the Chebyshev Acceleration method to the residual within the CG method; see Saad [12] for details.

Ashby et al. noted in [10] that an upper bound on the PCG condition number is given by:

$$\kappa_{pcg} = \frac{1+\epsilon_m}{1-\epsilon_m} \tag{4}$$

where

$$\epsilon_m \leq |T_m(\frac{\lambda_{max} + \lambda_{min}}{\lambda_{max} - \lambda_{min}})|^{-1}. \tag{5}$$

The total number of iterations, and hence sparse matrix-vector products, will be bounded from above by

$$k_{total} = \frac{\sqrt{\kappa_{cg}}}{2} \log_e(2/\epsilon) \tag{6}$$

where $\kappa_{cg} = \lambda_{max}/\lambda_{min}$, whilst the number of outer iterations, and hence dot products, will be bounded above by

$$k_{outer} = \frac{\sqrt{\kappa_{pcg}}}{2} \log_e(2/\epsilon). \tag{7}$$

Thus the ratio of $\sqrt{\kappa_{cg}/\kappa_{pcg}}$ gives us the approximate ratio of outer to inner iterations, and hence it provides a measure of the relative reduction in the number of global dot products in the CPPCG method compared to a more traditional PCG method. This reduction in global communication is the fundamental advantage of a CPPCG solver over other classes of CG-based sparse iterative solvers, and we will quantify its performance benefits later in this paper.

D. Selection of Parameters

The method is sensitive to the provision of accurate estimates of the extreme eigenvalues, which must be provided a priori. To estimate these eigenvalues, we perform several iterations of the regular CG method, before switching over to the CPPCG.

In choosing to use iterative (rather than direct) solvers we have made four choices, each less general than the last. These are (i) CG, (ii) Pre-conditioned Conjugate Gradient (PCG), (iii) Polynomially Preconditioned Conjugate Gradient (PPCG), and (iv) CPPCG. We have concentrated on using scaled and shifted first order Chebyshev polynomials exclusively for this work, and so we use the terms PPCG and CPPCG interchangeably for the rest of this paper.

IV. Many-core aware solvers

Though flat MPI has classically been a way to run massively parallel simulations, this approach has various problems when trying to take advantage of large-scale many-core systems. In particular, as many of today's supercomputers include heterogeneous architectures, it is necessary to expose data parallelism in an applications' computational kernels in order to exploit those resources.

In addition, halo data needed to store ghost information from other processes is duplicated in each MPI rank, and increasing the number of MPI ranks therefore increases the proportion of data that is duplicated. As halo depths are increased (up to 16 deep in our case), and the number of arrays storing data for each mesh point increases, this amount of redundant memory storage becomes a limiting factor on the number of data points that can be stored on each device.

The solution to this issue is to expose the maximum parallelism possible for the problem, which can then be exploited in a hierarchical manner at multiple levels within the node (vector, thread, task), instead of the traditional approach of a single MPI rank per CPU core. This approach should then give us the best chance to achieve good performance on a wide range of node architectures, from multi-core CPUs to heterogeneous CPU-GPU hybrid systems.

A. Parallelization

As TeaLeaf uses a matrix-free representation of the heat diffusion problem, each of the variables used can be stored as a two dimensional array (for the 2D case). The classical flat MPI structure decomposes the grid into rectangular subdomains, assigning each to an MPI rank, complete with halo cells for data exchanges. MPI ranks are then allocated one per CPU core. This structure is modified for the accelerated versions of TeaLeaf, by only assigning one (or possibly two, in the case of hybrid versions of TeaLeaf) MPI ranks per *node* (rather than one per core). With CPU core counts soon to approach 32 per socket, and heading towards $O(100)$ per node, this change alone provides a two orders of magnitude reduction in the number of MPI ranks required, which will improve TeaLeaf's long-term scalability on high node count machines.

Each of the computational kernels in TeaLeaf corresponds to one or more steps of the CG algorithm, and each employs a nested loop to iterate over the entire array (not counting boundary and halo cells). For example, part of the CG algorithm calculates the matrix A multiplied by a vector $\vec{p}$, then calculates the dot product of this result ($\vec{w}$) with $\vec{p}$:

$$\vec{w} = A\vec{p} \tag{8}$$

$$pw = \vec{p} \cdot \vec{w} \tag{9}$$

In Listing 1, Kx and Ky are the coefficients of the matrix A in the x and y dimensions of the grid. Each of the points on a 5 point stencil surrounding each cell is multiplied by one of these coefficients, which corresponds to a nonzero element in the sparse matrix A. The diagonal is one plus the sum of the rest of the nonzero elements on the row, making A diagonally dominant. As in this small code example, each of the grid points for all the other stages of the algorithm can be calculated independently. Using the old flat MPI model, each rank performs this operation on its own section of the grid before synchronizing with the other ranks in a global reduction of the pw value.

```
!$OMP PARALLEL REDUCTION(+:pw)
!$OMP DO
DO k=y_min,y_max
!$OMP SIMD
  DO j=x_min,x_max
    w(j, k) = (1.0_8                            &
      + (Ky(j, k+1) + Ky(j, k))                 &
      + (Kx(j+1, k) + Kx(j, k)))*p(j, k)        &
      - (Ky(j, k+1)*p(j, k+1)                   &
      + Ky(j, k)*p(j, k-1))                     &
      - (Kx(j+1, k)*p(j+1, k)                   &
      + Kx(j, k)*p(j-1, k))                     &

    pw = pw + p(j, k)*w(j, k)
  ENDDO
!$OMP END SIMD
ENDDO
!$OMP END DO
!$OMP END PARALLEL
```

Listing 1. Fortran/OpenMP implementation

This matrix-free sparse matrix vector multiplication, and the other computational kernels, expose sufficient independent work to lend themselves to parallelisation using the work sharing constructs in OpenMP. Using thread-level parallelism allows us to reduce the number of MPI ranks down to one per shared-memory node, improving the scalability problems seen at high node counts. In practice, we have observed marginally higher performance when running one MPI rank per NUMA node, avoiding memory transfers between the NUMA regions.

B. Many-core parallelization with OpenCL and CUDA

Due to their high memory bandwidth and highly parallel designs, GPUs are well suited to the kinds of computation required by sparse iterative solvers, which tend to be memory-bandwidth bound. OpenCL and CUDA are both examples of parallel programming models that can be used to target GPU devices. CUDA is specific to NVIDIA GPUs, while OpenCL is an open programming standard which is able to target GPUs from any vendor, as well as CPUs other forms of accelerator, such as FPGAs. Porting to these languages for a naturally parallel problem such as CG and its variants was relatively straightforward. As usual, data movement is the primary concern for optimal performance, and so the data is left resident in GPU memory during the solve, with only halos needing to be transferred between GPUs.

C. Minimizing communication

Efficient strong scaling is an important factor in the design of TeaLeaf's CPPCG solver. In this section we describe the two main approaches we used to help reduce the communication costs at larger scales.

1) Block Jacobi preconditioner: An effective preconditioner for TeaLeaf is the block Jacobi preconditioner, which splits the original matrix A into small blocks, each of which then have their own smaller preconditioning matrix, M. This approach results in a number of linear systems which can be solved in parallel, one for each block. We implement this approach in TeaLeaf by splitting the mesh into small 4×1 strips, with each strip corresponding to a small 4×4 block of the original matrix A. Because of the structure of the original matrix, these small blocks are tridiagonal, and can be solved trivially. Though there are parallel methods of solving tridiagonal matrices [13], these blocks are so small that it is in fact more computationally efficient to solve each of the small blocks in serial (the Thomas algorithm [14] is used in TeaLeaf, a much faster variation of Gaussian elimination for tridiagonal systems). Because TeaLeaf uses a matrix free representation of the problem, the preconditioner can also be effectively vectorized and threaded across blocks. As these blocks are independent, blocks at the edge of the boundaries of the mesh and at the boundary between neighboring processes are truncated into strips that are length 3, 2, or 1; these can be solved in the same fashion.

This block Jacobi preconditioner typically reduces the condition number of the matrix by around 40%, and has the advantage that it can be applied without any communication between neighboring processes.

2) Matrix powers kernel: The second approach taken with TeaLeaf's CPPCG solver to improve scaling is the use of the matrix powers kernel, often used with a variety of Krylov subspace-based solvers to avoid communication [15]. Typically the inner iterations of the CPPCG solver perform one matrix multiplication before requiring a halo exchange of data from neighboring processes to continue. The matrix powers kernel modifies this approach, exchanging a much deeper halo between neighboring processes and performing multiple matrix multiplications on this data, introducing a small amount of redundant computation in exchange for a significant reduction in communication. Figure 1 shows how the inner part of the CPPCG algorithm works when the halo depth is set to one. After one matrix multiplication – which, following the example in Listing 1, accesses data on a 5 point

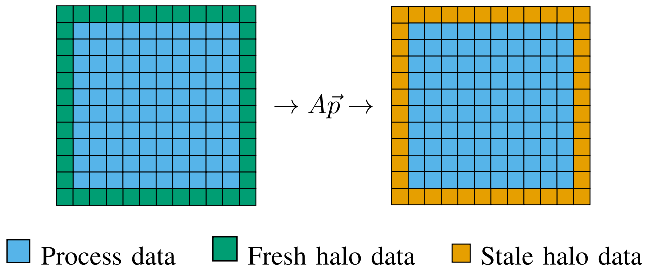

Fig. 1. Example of matrix multiplication on halo data. After one matrix multiplication, the halo data is stale and a halo exchange operation is needed.

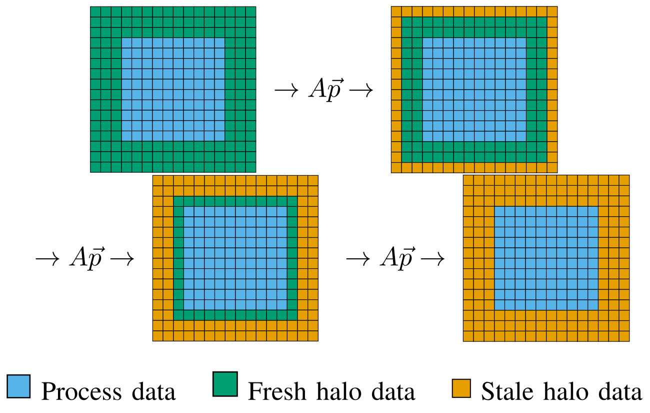
$\rightarrow A\vec{p} \rightarrow$
$\rightarrow A\vec{p} \rightarrow$
$\rightarrow A\vec{p} \rightarrow$
Process data
Fresh halo data
Stale halo data

Fig. 2. Matrix powers with a halo depth of three. Using this method, three matrix multiplications can be performed before a halo exchange is required.

stencil – the halo data is stale. At this point, another halo exchange needs to be performed to get fresh data (in reality, the corner data is not used, but it is shown here for ease of understanding). In contrast, Figure 2 shows how the matrix powers kernel works with a matrix powers halo depth of three. In this case, the matrix multiplication kernel we run is similar to Listing 1 but the loop bounds are extended to include the halo data that has been exchanged from neighboring processes. This means that multiple processes will perform the matrix multiplication operation on overlapping data, introducing a small amount of redundant work into the calculation. In reality, as the code is well vectorized and threaded, this overhead is negligible as long as the halo is kept to an appropriate depth. Experiments have shown that this is typically no more than 8 for CPUs and 16 for GPUs.

After the matrix multiplication kernel has been run once, the outer layer of the halo data is then stale. Some other vector-vector operations are then performed, using the same 'extended' bounds as with the matrix multiplication kernel, then the loop bounds are moved in by one cell and another matrix multiplication can be performed. This process is repeated until all of the halo data is stale, which forces a halo exchange, but this occurs much less frequently than with the standard halo depth of one. This approach causes more data to be exchange with neighboring processes at each halo exchange, but in general this change also aids performance: with a matrix powers kernel halo depth of n, we communicate approximately n times as much data at halo exchange, but we do this n times less frequently, so the total amount of data communicated will be the same while messages become larger.

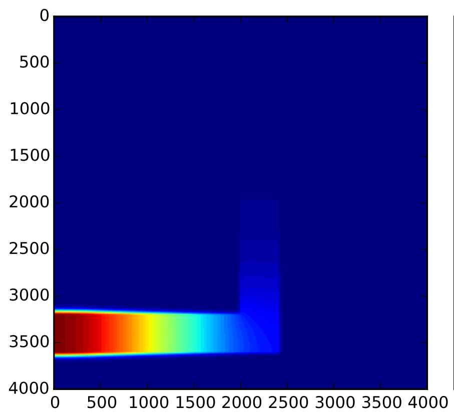

Fig. 3. TeaLeaf 'crooked pipe' 4000x4000 domain after 15 microseconds. Redder colors indicate higher temperatures.

One side effect of the matrix powers kernel is that because the bounds of the area of the mesh being computed are constantly changing, the block Jacobi preconditioner cannot be used. Because the block preconditioner relies on up-to-date values of the whole block (in this case, the 4x1 strip of data), this would require exchanging with neighbouring processes on each iteration, eliminating the benefit of reduced communication provided by the matrix powers kernel.

V. Experimental Setup

A. Machines and Software Versions

Table I shows the setup of the systems used for benchmarking TeaLeaf's CPPCG solver against a baseline CG solver and also PETSc's CG solver coupled with Hypre's BoomerAMG as a preconditioner.

On both Piz Daint and Titan the Cray MPI library was used. On Spruce we used SGI MPT 2.11. We also used third party libraries on Spruce, including SGI's modified version of PETSc which was based on version 3.5.3, and Hypre version 2.10b (the most current version at the time of our experiments) for its BoomerAMG preconditioner. Note that our Piz Daint results were gathered before its recent upgrade to NVIDIA P100 GPUs.

B. Heat Diffusion Test Case

The data set used for our experiments simulates a dense material of low heat conduction. A crooked pipe of lower density material that has a higher heat conduction passes through the dense material, traversing from one side of the problem domain to the other with a number of kinks. A fixed time step of 0.04 microseconds is used throughout each numerical experiment. Figure 3 shows the average temperature across the domain at the end time of 15 microseconds, plotted against mesh resolution. Blue areas in the graph represent colder areas, with the redder colors representing hotter areas. As the 'pipe' part of the domain has a lower density, heat travels faster along this area than elsewhere in the domain.

System	Spruce	Piz Daint	Titan
Compute device	E5-2680v2	NVIDIA K20x	NVIDIA K20x
Total cores	40,080	115,984	560,640
Interconnect	SGI Altix ICE-X	Cray Aries	Cray Gemini
Driver/compiler versions	Intel 15.0	340.87 (CUDA 6.5)	352.101 (CUDA 7.5)

TABLE I
TEST SETUP SPECIFICATIONS

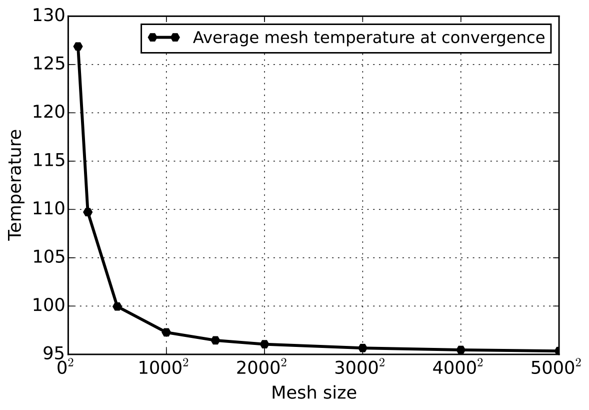

Fig. 4. Convergence of temperature as mesh size increases.

The real-world problems that motivated this work all have a maximum interesting mesh size of around 4000x4000, as shown in Figure 4 – as the size of the mesh increases, the average temperature that the mesh converges to stops changing, with this 4000x4000 test case being the point at which any further resolution increase becomes less scientifically interesting. For this reason, this study will concentrate on strong scaling of mesh converged calculations of 4000x4000.

VI. Strong Scaling Results

As per the prior discussion regarding mesh convergence, strong scaling is more relevant than weak scaling for our real-world problems. Weak scaling performance would also be more difficult to characterize: the nature of the algorithm means that increasing the mesh size also increases the condition number, the number of iterations required to converge, and hence the time to solution. For these reasons, only strong scaling results are presented.

In the following graphs, each line is labeled with the solver used as well as the halo depth. For example, "PPCG - 1" refers to using the CPPCG solver with a halo depth of 1, while "PPCG - 16" uses a halo depth of 16.

Figure 5 shows strong scaling results for the CUDA version of TeaLeaf, across up to 8,192 GPUs (nodes) of Titan. The best CUDA implementation (PPCG 16) achieves a time of 4.26 seconds at 8,192 nodes. It should be noted that TeaLeaf scaling plateaued once we reached 1,024 nodes on Titan.

Figure 6 shows strong scaling results for the CUDA version of TeaLeaf across up to 2,048 GPUs (nodes) of Piz Daint. At 2,048 nodes on Piz Daint, the CUDA version ran in 2.79 seconds and on Titan the same problem on the same number

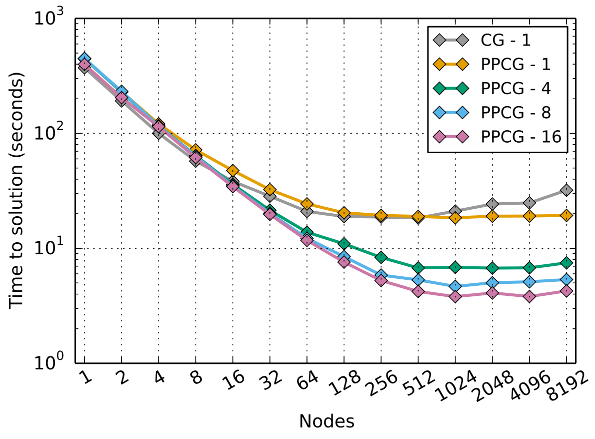

Fig. 5. CUDA strong scaling on Titan.

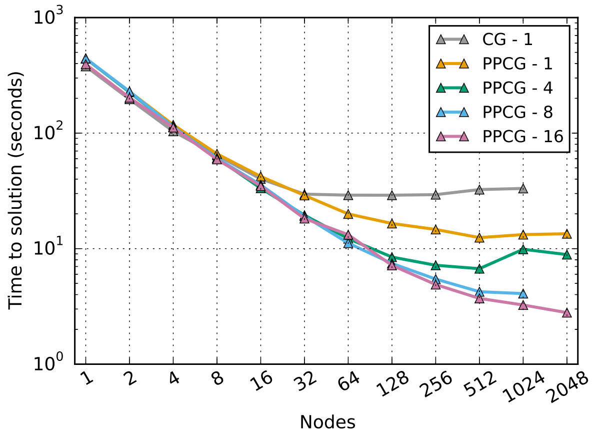

Fig. 6. CUDA strong scaling on Piz Daint.

of nodes (GPUs) ran in 4.09 seconds. As both systems were using the same GPUs, this 47% strong scaling performance improvement can be attributed to the fully connected network on Piz Daint, despite newer drivers on Titan.

One can see from the results on both Titan and Piz Daint that, as predicted, the CPPCG method strong scales significantly better than CG. The benefits of the matrix powers kernel approach is also clear, with improvements in performance still increasing at halo depths of 16 on both systems. The strong scaling nature of the fixed 4000x4000 problem size starts to show after 1,024 nodes on Titan. After this point, adding more nodes actually increases the wall-clock time for the solver. With just (4000x4000/1024) ≈ 15,625 grid points

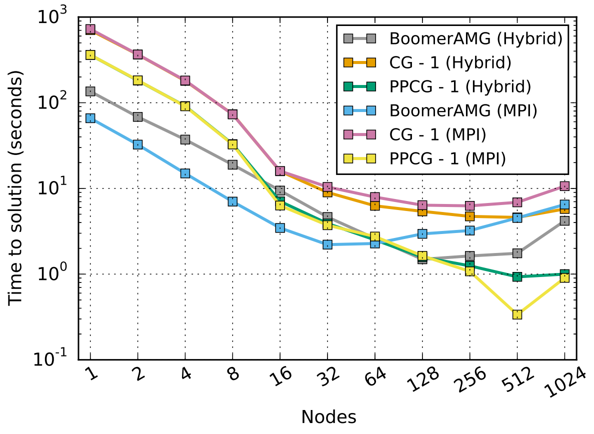

Fig. 7. MPI and Hybrid strong scaling on Spruce.

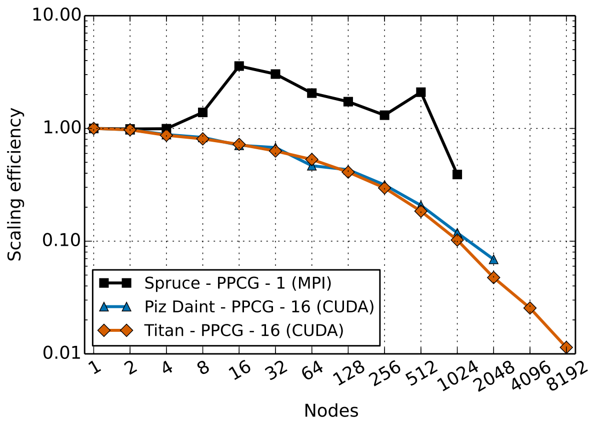

Fig. 8. Comparison of scaling efficiency across different test systems.

per GPU at 1,024 nodes, at this scale we have barely four grid points per processor element (PE) in each GPU, and thus it is not unexpected for the knee in the curve to be at around 1k nodes. Figure 7 shows the scaling of both the flat MPI and the OpenMP/MPI hybrid versions of TeaLeaf's CG and CPPCG solvers on Spruce. As Spruce is a purely CPU-based system, we were also able to compare the performance of TeaLeaf's CG-based solvers with PETSc's CG solver coupled to the BoomerAMG preconditioner [6], [16], [17]; this configuration is referred to as "BoomerAMG" in the graph.

Due to available time constraints on Spruce, only the results for a halo depth of 1 were gathered. As testing on the GPU based systems shows, increasing the matrix powers halo depth can result in much better scaling for the CPPCG solver. Unlike the GPU based systems where scaling is still improving even up to a depth of 16, preliminary testing shows that this benefit plateaus at around 8 on CPU based systems when the amount of redundant computation starts to outweigh the benefit of reduced communication.

The PETSc CG with BoomerAMG preconditioner implementation is the fastest at low node counts (1-8 for hybrid, 1-64 for flat MPI), while our CPPCG solver's communication avoiding approach provides greater strong scaling capability from 128 nodes onwards. The PETSc+BoomerAMG's strong scaling performance peaks at just 32 nodes, with slower performance delivered when adding nodes beyond 32. TeaLeaf's CPPCG solver continues to improve in performance all the way up to 512 nodes before it peaks, with its hybrid and flat MPI versions delivering near identical performance at all scales. At 512 nodes the CPPCG implementation delivers twice the performance of the best PETSc+BoomerAMG configuration at that scale. Increasing the CPPCG halo depth is expected to improve both its scaling and performance further.

Figure 8 is a comparison of the *scaling efficiency* of the best implementation across all systems. As mentioned, the scaling results for CPPCG were only gathered for a halo depth of 1 on Spruce, but at the number of nodes that results were collected for, cache effects mean that the scaling of the MPI version (as well as the hybrid version) of the CPPCG solver maintains super linear scaling up to 512 nodes, beating both Piz Daint and Titan in terms of both time to solution and scaling efficiency. Finally, as previously mentioned, the scaling on Piz Daint is consistently higher than Titan on higher node counts due to the higher performance of Piz Daint's fully configured Cray Aries interconnect compared to Titan's previous generation Cray Gemini interconnect.

VII. Impact and Future Work

In this work, TeaLeaf has proven its effectiveness in exploring the design space of CG-based solvers. TeaLeaf has also proven useful as a way of comparing parallel programming languages, with ports to Kokkos [18], RAJA [19], OpenMP 4.5 [20] and OpenACC [21], complementing the MPI, OpenMP 3, CUDA and OpenCL ports described in this paper. Results of these comparisons have appeared in numerous papers [22], [23], [24], [25], [26].

Going beyond CPPCG, we intend to explore combining the favorable aspects of both domain decomposition and agglomeration multi-grid methods, based on solver components with favorable performance characteristics. Using deflation techniques [27] we will be able to represent these low energy modes in a series of nested lower dimensional sub-spaces. This multi-level approach should improve the weak scaling behavior of the solver, leading to convergence behavior closer to a true multi-grid scheme, but with improved computational performance from exploiting the properties of the underlying hardware. We also plan to investigate short term gains which can be obtained from potential improvements to the implementation of existing algorithms. The Krylov solver can be restructured so that the multiple dot products are combined into a single communication step and the communications can be overlapped with the application of the preconditioner. More effective preconditioning strategies will be explored such as approximate incomplete factorizations [28] and similar approaches based solely on matrix-vector products.

VIII. Conclusions

We have shown that a carefully constructed mini-app such as TeaLeaf can prove very effective in the design space exploration for future sparse iterative solvers. Using TeaLeaf we have already gathered some useful insights. At low node counts, third party iterative solvers that are best in class, such as PETSc's CG combined with BoomerAMG as a preconditioner, perform well. These methods are also robust for systems with large condition numbers. However, these solvers struggle to perform well when strong scaling up into the Petascale regime, where the simpler (non-AMG) methods start to outperform them. The AMG solvers are also not mature or widely available for accelerated technologies and the set up cost for the nested operators is expensive. The CPPCG solver presented in this paper directly addresses these issues.

Mapping CG and CPPCG to accelerators is relatively straightforward and has given an initial capability that strong-scales well on these types of machines. Whether these simpler methods can cope with extreme condition numbers robustly is an open question. It is known that a purely Chebyshev-based solver cannot cope with some condition numbers that we are interested in, but they do function well as a smoother or preconditioner. We believe that we will need to keep developing methods even beyond CPPCG which continue to minimize communications and maximize concurrency while using methods from the deflation and AMG space that will enhance robustness at scale without losing performance. TeaLeaf has already proven itself an invaluable tool for these solver design space explorations.

TeaLeaf's source code is freely available to download from Github (http://uk-mac.github.io/TeaLeaf/)

Acknowledgments

Much of this work would not have been possible without the support and advice of Intel, AMD, Cray, PGI and NVIDIA. We are also grateful for access to the resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725, and for access to the resources of the Swiss National Supercomputing Centre (CSCS). We also acknowledge the use of the ACRC Blue Crystal supercomputer facilities, and the HPC Zoo at the University of Bristol.

References

[1] J. Murray, "Mathematical biology," *C271*, 1989.

[2] S. Chapman and T. G. Cowling, *The mathematical theory of non-uniform gases: an account of the kinetic theory of viscosity, thermal conduction and diffusion in gases*. Cambridge University Press, 1970.

[3] R. L. Bowers and J. R. Wilson, "Numerical modeling in applied physics and astrophysics," *Boston: Jones and Bartlett*, 1991.

[4] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. Beckingsale, A. Mallinson, and S. Jarvis, "Accelerating hydrocodes with OpenACC, OpenCL and CUDA," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, Nov 2012, pp. 465–471.

[5] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep. SAND2009-5574*, 2009.

[6] S. B. et al, "PETSc Web page," http://www.mcs.anl.gov/petsc, 2015. [Online]. Available: http://www.mcs.anl.gov/petsc

[7] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the Trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.

[8] R. Falgout and U. Yang, "hypre: A library of high performance preconditioners," *Computational ScienceICCS 2002*, pp. 632–641, 2002.

[9] D. P. O'Leary, "Yet another polynomial preconditioner for the conjugate gradient algorithm," *Linear Algebra and its Applications*, pp. 377–388, 1991.

[10] S. F. Ashby, T. A. Manteuffel, and J. S.Otto, "A comparison of adaptive Chebyshev and least squares polynomial preconditioning for Hermitian positive definite linear systems," *SIAM J. Sci. Stat. Comput.*, vol. 13, no. 1, pp. 1–29, January 1992.

[11] B. N. Parlett, *The Symmetric Eigenvalue Problem*. SIAM Classics in Applied Matehmatics, 1998.

[12] Y. Saad, *Iterative Methods for Sparse Linear Systems*. SIAM, 2003.

[13] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the GPU," *ACM SIGPLAN Notices*, vol. 45, no. 1, p. 127, 2010.

[14] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. John Hopkins University Press, 1996.

[15] M. Hoemmen, "Communication-avoiding Krylov subspace methods," Ph.D. dissertation, University of California, Berkeley, 2010.

[16] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*. Birkhäuser Press, 1997, pp. 163–202.

[17] V. E. Henson and U. M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155 – 177, 2002, developments and Trends in Iterative Methods for Large Systems of Equations - in memorium Rudiger Weiss. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0168927401001155

[18] H. C. Edwards and C. R. Trott, "Kokkos: Enabling performance portability across manycore architectures," in *Extreme Scaling Workshop (XSW), 2013*. IEEE, 2013, pp. 18–24.

[19] R. Hornung, J. Keasler *et al.*, "The RAJA Portability Layer: Overview and Status," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-661403, 2014.

[20] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 4.5," 2015.

[21] OpenACC Steering Committee, "OpenACC Application Program Interface Version 2.5," 2017.

[22] M. Martineau, S. McIntosh-Smith, M. Boulton, and W. Gaudin, "An evaluation of emerging many-core parallel programming models," in *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, 2016, pp. 1–10.

[23] M. Martineau, S. McIntosh-Smith, and W. Gaudin, "Evaluating OpenMP 4.0's effectiveness as a heterogeneous parallel programming model," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 338–347.

[24] M. Martineau, S. McIntosh-Smith, M. Boulton, W. Gaudin, and D. Beckingsale, "A performance evaluation of Kokkos & RAJA using the TeaLeaf mini-app," in *The International Conference for High Performance Computing, Networking, Storage and Analysis, SC15*, 2015.

[25] M. Martineau, J. Price, S. McIntosh-Smith, and W. Gaudin, "Pragmatic performance portability with OpenMP 4.x," in *International Workshop on OpenMP*. Springer, 2016, pp. 253–267.

[26] M. Martineau, S. McIntosh-Smith, C. Bertolli, A. C. Jacob, S. F. Antao, A. Eichenberger, G.-T. Bercea, T. Chen, T. Jin, K. O'Brien *et al.*, "Performance analysis and optimization of Clang's OpenMP 4.5 GPU support," in *Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), International Workshop on*. IEEE, 2016, pp. 54–64.

[27] J. Frank and C. Vuik, "On the construction of deflation-based preconditioners," *SIAM Journal on Scientific Computing*, vol. 23, pp. 442–462, 2001.

[28] E. Chow and A. Patel, "Fine-grained parallel incomplete LU factorization," *SIAM Journal on Scientific Computing*, vol. 37, pp. C169–C193, 2015.