ME 469 Report - Parallel `mycavity`

Pavan B. Govindaraju, Emeric S. Boigné

March 26, 2017

Abstract

The methodology and implementation details for successfully parallelizing the starter `mycavity` code using CPUs and GPUs is presented. Profiling of the starter code is performed and it is demonstrated that the linear solver is most amenable to parallelization. Both direct and iterative solvers are considered for the problem at hand. The `Armadillo` C++ library is utilized for the sequential solver study and iterative solvers are shown to be favorable. Parallel solvers are then discussed along with implementation in the current code using `Elemental`, a state-of-the-art C++ linear algebra library. Runtime reduction is observed upto 2048 cores and scaling studies performed on the multi-CPU version show favorable weak and strong scaling. The GPU version of the code uses `PARALUTION` and is then discussed in detail, along with comparison of various iterative solvers. Speedup of upto a factor of 3 on porting the linear solver to the GPU is observed.

Contents

1 Introduction

The evolution of multi-core processing units towards many-core units, as well as the dramatic increase of the number of cores in super-computers enable the tackling of new multi-physics problems. However, this great computing power may be utilized only with suitable softwares and requires additional effort compared to writing conventional software. The starter `mycavity` code is an elementary CFD solver, which operates on structured grids, but given that it also solves the temperature equation, it can be placed above many other codes in terms of complexity and would benefit from leveraging better computing infrastructure. This manuscript discusses the efforts towards parallelizing the `mycavity` code.

The first section discusses the code profile and call graph. This is followed by a discussion on linear solvers and appropriate algorithms for the current problem. The approach for code optimization focuses on the linear solver and is split into sequential and parallel solvers, and is the subject of discussion for the two following sections. Details regarding implementation and verification are presented, along with numerical and computational results. The final section offers conclusions and a short note on future directions for this work.

1.1 Code profiling

The first step towards determining the routines which are needed to be optimized or parallelized is to 'profile' the code. Care was taken to ensure that the starter code was compiled using optimization flags, which in this case was `'-O3'`. Addditionally, the debugging flag `'-g'` is enabled for the profiler. The code was profiled using `callgrind` and the results were visualized using `qcachegrind`. Once the program is compiled, the following commands are run in order

```
>> g++ -O3 -g mycavity.cpp
```

```
>> valgrind --tool=callgrind ./a.out
```

This produces a file `callgrind.out.XXXX`, which can be open with `qcachegrind` using

```
>> qcachegrind callgrind.out.XXXX
```

There are two main visualizations for code performance and are referred to as

1. Call graph : Represents calling relationships between subroutines in a program. Also augmented with proportion of runtime taken by certain subroutines.

2. Code profile : Detailed map containing the resource and execution time splits for each subroutine in a program.

The call graph for the starter code is shown in Figure 1. The number of calls for each subroutine are shown and are dependent on the input parameters. In this case, 10 iterations were provided per timestep and a total of 51 timesteps were performed,

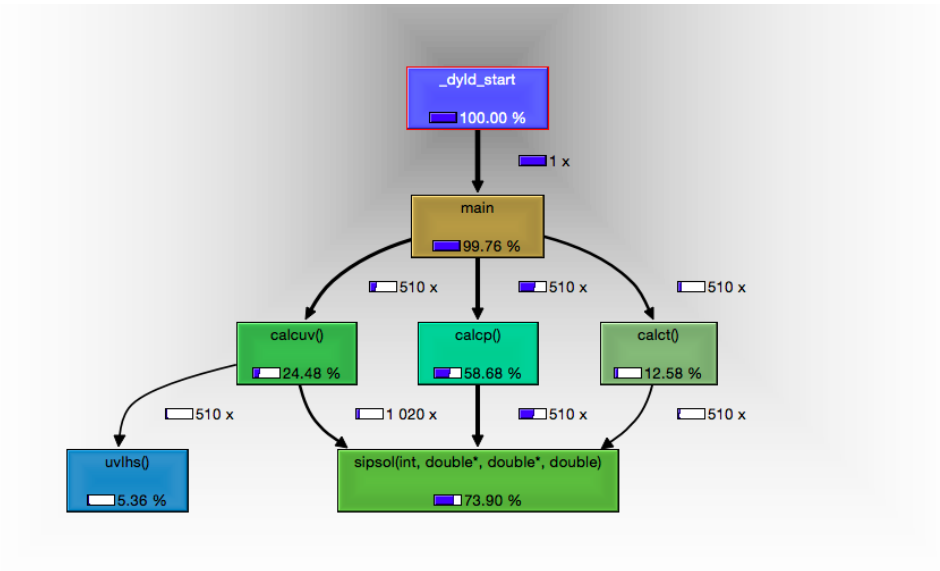

Figure 1: Call graph of starter `mycavity` code

thus leading to 510 calls for most functions. The linear solver function goes by the name `sipsol()` and is invoked twice in the function `calcuv()`, thus leading to 1020 function calls. The important thing to read from this figure is that around 3/4ths of the program execution time is spent in the linear solver and thus, the best way to optimize the code is to direct efforts towards speeding up the linear solver step. Further information regarding the proportion of time spent in the solution of the linear system within each subroutine of the code is shown by the code profile.

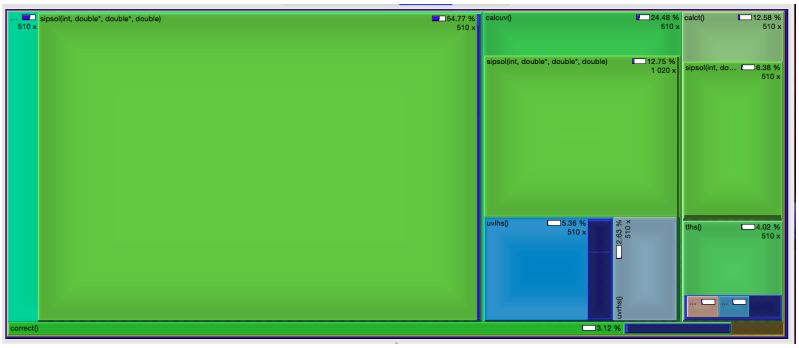

Figure 2: Code profile of starter `mycavity` code

Multiple inferences regarding the particular matrices being solved inside each subroutine can be obtained from Figure 2 based on the fact that despite similar dimensions, there is a vast difference in the amount of time `sipsol()` takes within each subroutine. This is however not the focus of this discussion, but it is important to note the structure of the matrices at each stage so that an appropriate choice for the linear solver can be made.

1.2 Linear solvers

This section discusses the linear solvers which are suitable for usage in the `mycavity` code. Both sequential and parallel solvers have been considered in this manuscript along with a comparison between direct and iterative solvers for the particular problem. The parallelization strategy conveniently splits into two tasks, with one member of the team handling CPU implementations and the other harnessing GPUs to solve this problem. Given that a major chunk of the program involves floating point operations, utilizing GPUs for the linear solve is a logical proposition. It is important to note that the matrices arising from the PDE discretization are all sparse.

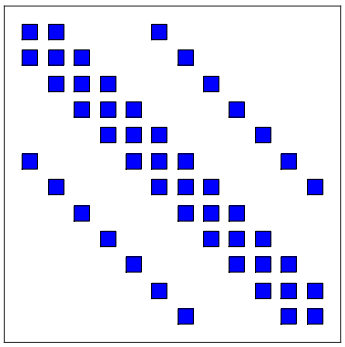

Figure 3: Sparsity pattern of matrices in `mycavity`

Figure 3 shows the sparsity pattern of matrices arising in the code. Each non-zero term of the matrix represents fluxes in the North, South, East or West directions. The separation between the terms along the diagonal and the separated off-diagonals is governed by the grid size. Thus, the given matrix does not fit the conventional description of a banded matrix and solvers designed particularly for that purpose cannot be applied.

The algorithm used in the starter code deserves special mention and is borrowed from [1]. A summary of the procedure is also presented in [2] and has thus not been reproduced in this report. It is an example of a *sequential*, *iterative* solver and leaves open the comparison against the other alternative, *direct* solvers. They have certain advantages and can be summarized as

- Same result if the procedure converges, irrespective of the algorithm
- Speedup by large factors for problems involving multiple right hand sides
- Relatively more robust when good preconditioners are not available

On the other hand, the most commonly cited disadvantage for direct solvers is memory usage. This can be circumvented through the usage of *distributed* solvers, which utilize multiple memory locations for storage of matrices. Some solvers also perform *out-of-core*, which means that the solution procedure utilizes disk memory. As memory resources were not the constraint, especially for lower resolutions, the usage of a direct solver was strongly considered. Since the matrices were also known to be sparse, the first candidate for the replacement is a *sparse*, *direct* solver and to keep things simple, *sequential* solvers were initially considered.

2 Sequential solvers

This subsection discusses the usage of sequential solvers for solving the linear systems arising within the code. Based on the discussion in Section 1.2, sparse-direct solvers are considered and the first non-trivial obstacle for this approach involves handling sparse matrices. This is circumvented through the use of support routines as the storage and access are not done in a straightforward manner. Details regarding the underlying techniques for sparse matrix frameworks are described in [3]. The sparse-direct solver chosen is `SuperLU`, which is a well-established modification of sparse Gaussian elimination and leverages memory hierarchies within computers. An in-depth discussion of the algorithm can be found in [4].

2.1 Implementation

The current report uses `Armadillo` - a C++ linear algebra library [5], which has excellent support for sparse matrices and thus gives the flexibility to change the solver with minimal modification. Also, `SuperLU` was installed as one of the default solvers and the linear solve step could directly incorporate it with a mere change of arguments. In fact, this flexibility was recognized a priori and adding to the fact that `Armadillo` was also based in C++ made it the obvious choice for support matrix routines. The end result is that the `sipsol()` call is replaced by `sipsol_arma()`, which encapsulates the linear problem construction, solution and re-population of the physical variables.

2.2 Verification

The verification procedure employed for the sequential program can be briefly summarized as

- Matrix : Ensuring the correct sparse matrix structure is obtained
- Linear solver : The solution of matrix inversion is compared with that obtained using Stone's implicit method
- Reassignment : This is easier to verify once the correct matrix is generated. If on subsequent timesteps, the generated matrix is the same using both methods, then the reassignment must be correct.
- Field quantities : The quantities of interest which are output in `mycavity.out` are then compared as a final check.

2.3 Results

The first three steps presented in Section 2.2 have been verified to machine precision for the `Armadillo`-based version of the code. Note that the first timestep for the pressure solver has been performed using `sipsol()` as the condition number (κ^{-1} = 7.02e-18) is too high for a direct solver. The verification was first carried out on a simple 10x10 grid and default input parameters for the quantities of interest.
The agreement of both methods is summarized in Table 1. Once this was established, runtime benchmarks were performed. This was carried out for a 100x100 grid and

Method	Time	West Heat Flux	East Heat Flux	Maximum velocity
`Standard`	0.0	3.449262	-3.456144	164.516248
	1.0	-4.036398	-2.481468	12.022349
	2.0	-3.508961	-3.188939	9.520496
	3.0	-3.380951	-3.320071	9.385856
	4.0	-3.356270	-3.344737	9.365512
	5.0	-3.351594	-3.349409	9.361662
`Armadillo`	0.0	3.408158	-3.416660	162.702922
	1.0	-4.065739	-2.438924	12.753565
	2.0	-3.516220	-3.180282	9.539231
	3.0	-3.382508	-3.318489	9.387167
	4.0	-3.356567	-3.344440	9.365753
	5.0	-3.351651	-3.349353	9.361708

Table 1: Convergence study comparing the Stone's procedure (standard) and `Armadillo`-based implementation

in addition to the previous algorithms, the test was also performed using a *dense*, *direct* solver - `LAPACK` [6]. This additional test also helps to highlight the advantage of using the sparse structure of the matrices in the current problem. The timings are listed in the case of a 100x100 grid, averaged over 10 solutions for the u-component in the `calcuv()` routine.

- SIP : 0.708s
- `SuperLU` : 2.91s
- `LAPACK` : 84.2s

It is noted that the default iterative solver outperforms the implemented sparse-direct solver. Given that the iterative solver is optimized for solving matrices arising from PDE discretization, this conclusion should not come as too much of a surprise. However, this sets the stage for eventual work on parallel solvers, where emphasis is placed on iterative solvers, given the nature of the problem. The dense solver is obviously outperformed and also comes with the added disadvantage that the roundoff error now increases due to unrestricted fill-in.

3 Parallel solvers

This section discusses the usage of parallel solvers for solving the linear systems arising within the code. This is done either through the usage of multiple processors or through the usage of GPUs. Both approaches are motivated by the fact that solutions to linear equations involves a vast number of floating point operations for a given amount of data transfer, thus making them *compute* limited.

3.1 Multi-CPU

3.1.1 Implementation

The Multi-CPU implementation is performed using `Elemental` [7] - a state-of-the-art C++ linear algebra library, which uses MPI calls for the distribution of data. In addition, several support routines are part of the library and help simplify the handling of distributed sparse matrices. The in-built distributed sparse linear solver uses modified GMRES along with a Cholesky-like factorization to construct the preconditioner, details of which are available in [8]. The fact that `calcp()` gives rise to a symmetric matrix is utilized and the in-built Hermitian solver, specified by `HermitianSolve()`, is employed in this case. Note that this uses a sparse-direct (Bunch-Kaufman [9]) algorithm and is preferred because it is established that the `HermitianSolve` routine in `Elemental` is atleast as stable and fast as the general iterative solver, in case of symmetric or Hermitian matrices.

3.1.2 Verification

In addition to the steps outlined in Section 2.2, a distributed version of the program presents additional obstacles due to the parallelism and requires additional checks. These can be briefly summarized as

- Each of the steps presented must be reproducible not just when multiple computing cores are used, but also when multiple computing nodes are utilized.
- The output must be consistent between multiple runs and even the runtime should roughly be the same. This ensures the lack of race conditions or an excessive dependence on a particular processor to finish its task, which in parallel programming parlance, are referred to as bottlenecks.

3.1.3 Results

The verification benchmarks were performed using three processors, each on a different computing node. This is in line with the recommendations made in Section 3.1.2. In addition, the steps listed in Section 2.2 were also performed using multiple computing nodes and cores. One important comment is that both the sparse-direct and iterative solver in `Elemental` are robust enough to handle `calcp()` in the first timestep and thus, the same kind of solver is utilized independent of time. The convergence verification carried out using a 10x10 grid and default input parameters for the quantities of interest, which can be summarized as

The agreement of both methods is summarized in Table 2. Once this was established, runtime benchmarks were performed and given the parallel nature of this implementation, several interesting studies regarding the performance of the program could be undertaken. The first test was carried out on 192 cores for a 1000x1000 grid with default input parameters. Runtimes for each substep were noted and the reported values are based on averages for 10 timesteps. These can be listed as

- Linear solve : 5.1s
- Symmetric solve : 5.8s
- Repopulate entries : 1.4s

Method	Time	West Heat Flux	East Heat Flux	Maximum velocity
`Standard`	0.0	3.449262	-3.456144	164.516248
	1.0	-4.036398	-2.481468	12.022349
	2.0	-3.508961	-3.188939	9.520496
	3.0	-3.380951	-3.320071	9.385856
	4.0	-3.356270	-3.344737	9.365512
	5.0	-3.351594	-3.349409	9.361662
`Elemental`	0.0	3.398592	-3.398592	151.922695
	1.0	-4.067705	-2.439657	12.753565
	2.0	-3.513874	-3.184667	9.524109
	3.0	-3.381790	-3.319252	9.386476
	4.0	-3.356422	-3.344586	9.365630
	5.0	-3.351622	-3.349382	9.361684

Table 2: Convergence study comparing the Stone's procedure (standard) and `Elemental`-based implementation, run using 3 cores and separate nodes

In addition, the `Elemental`-based code was run on a 100x100 grid on a single processor to provide a comparison against results in Section 2.3. The average time for one solution is 0.431s, which is lesser than the time taken by Stone's procedure. The next test involved providing the runtime estimate of each important subroutine in the CFD solver again for the 1000x1000 case. The results can be summarized as

- `calcuv()` : 13.6s
- `calcp()` : 7.4s
- `calct()` : 6.8s

Given that the parallelization ability of the code was well-established, further understanding regarding the scaling of this program was required. A 4000x4000 grid was chosen, irrespective of the physical requirement of the problem, and the time for each iteration in a timestep was performed. Due to resource constraints, the test was performed only upto a maximum of 2048 cores and the runtime was still found to decrease with an increase in the number of processors.
The successful scaling of the program upto 2048 cores is demonstrated in Figure 4. Note that only a decrease in runtime with more number of processors is claimed and comments about efficiency require a more careful evaluation. This prompted a study on weak and strong scaling of the program at hand. It is useful to recall the definitions of weak and strong scaling at this stage. Both studies look at the variation in solution time with number of processors for

- Weak scaling : fixed problem size per processor.
- Strong scaling : fixed total problem size

The problem size is roughly found to scale as $O(n^3)$, where n is the number of grid points in each dimension. Thus, to maintain a fixed problem size per processor, the

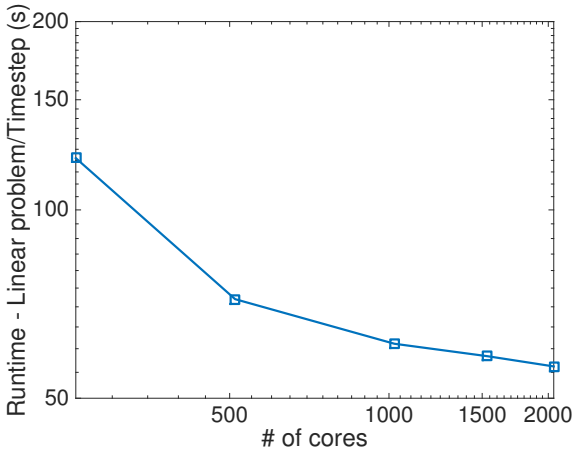

Figure 4: Runtime per iteration (linear solve + repopulate) in a timestep for 4000x4000 grid, averaged over 5 calls as a function of number of computing cores for the `Elemental`-based code

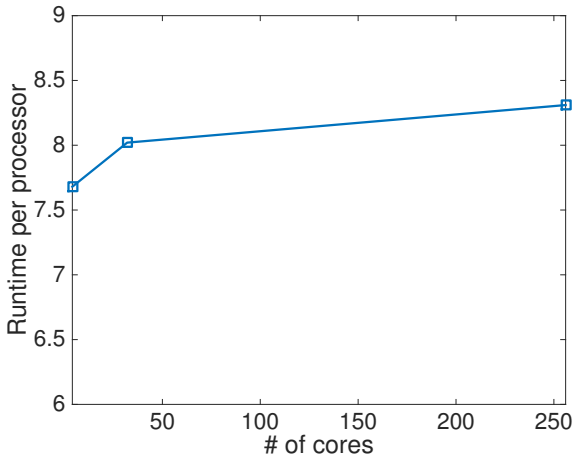

Figure 5: Weak scaling in solution time averaged over 5 calls as a function of number of computing cores for the `Elemental`-based code. The problem size was chosen to be 160x160 for the 256 cores case.

number of cores used is increased eightfold on doubling the number of grid points per dimension.

Weak scaling is verified upto 256 cores and is shown in Figure 5. Note that the communication is already starting to show some effect and thus highlights how weak scaling studies are difficult, in the sense that maintaining a fixed load per processor while scaling the problem is difficult. The next logical step is to perform a strong scaling analysis, which is easier to perform, as the problem size is kept fixed.

Though far from the ideal profile, Figure 6 demonstrates that the current code is capable of nominal strong scaling. Further work would involve parallelizing the flux computation and minimizing the communication overhead required.

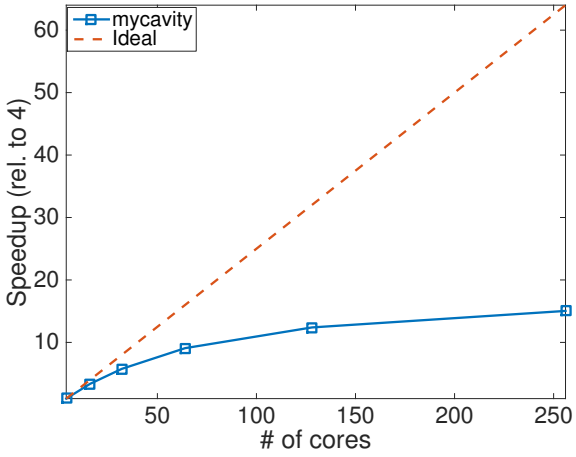

Figure 6: Strong scaling in solution time for 1000x1000 grid, averaged over 5 calls as a function of number of computing cores for the `Elemental`-based code

3.2 GPU

3.2.1 Implementation

The GPU implementation was performed using the C++ sparse iterative solver and preconditioner library `PARALUTION`[10]. This library features an all inclusive dynamic framework providing sparse matrix iterative solvers and preconditioners implementations for multi-core CPU and GPU devices. The `sipsol()` function from the `mycavity` code is implemented using this framework. Several solvers were developed and compared, both for CPU and GPU. The GPU computations were performed on a Tesla C2070 using CUDA 7.0 on the Rye-1 Farmshare Stanford cluster. In order to accommodate for proper comparisons between the CPU and GPU implementations, the `PARALUTION` source code was modified to enable a more relevant determination of the residuals. These modifications should be reproduced in order to run the GPU version of the `mycavity` code. Further description of these modifications is provided in Appendix A.

Due to the symmetric nature of the Poisson pressure system, two different solvers were implemented: one to solve the general (non-symmetric) systems for the velocity field and temperature, and one to solve for the symmetric pressure system. Solving the Poisson equations for large systems is the most expensive step and making use of the symmetry of the matrix decreases the computation time. The original CPU Stone's method solver doesn't make use of this property and was thus improvized upon both in the multi-CPU and GPU implementations. The comparisons with the original CPU solver were limited to grid smaller than 100,000 elements.

3.2.2 Verification

The `PARALUTION` implementation was verified by performing a direct comparison with the CPU results obtained using the original Stone's Method solver for a grid of moderate size. Due to the iterative nature of the solvers involved, a proper comparison requires setting up sufficient tolerance requirements so that both solvers converge to a similar solution. To this end, it is important to recall that the `mycavity` code involves two scales of iterative loops. The inner one concerns the linear solver

`sipsol()` and is called for each computation of the velocity field, pressure correction and temperature. An outer loop is performed over each series of these measurements at each timestep. In order to converge to a common field, relative tolerance requirements were set to 10^{-6} for the inner solvers, and to 10^{-5} for outer ones. The number of iterations was increased up to 3000 for the Poisson solver (inner loop).

The `PARALUTION` computations match the results obtained by the original code accurately for small grids (only 0.8 % of a comparison norm for a uniform mesh of 128x128 elements). As the grid size is increased, the difference between the results increases slightly, but remains moderate. Even for the higher discrepancy observed (10 % of deviation for a uniform mesh of 256x256 elements), visual comparisons of the flow field showed reasonable agreement.

It was also verified that the CPU and GPU versions of the `PARALUTION` solvers obtained similar results. Agreement was reached down to machine error even for large size grid in this case.

3.2.3 Results

Benchmarks compared the computation time for grids of increasing size for the `PARALUTION` CPU and GPU implementations, and the original CPU Stone's method solver. In order to investigate the reduction provided by the GPU implementation at larger scale, only the first timestep was computed in the following results. This choice was motivated by the observation that for small grids, the computation cost doesn't vary much between each timestep and for each iteration.

Comparisons between the `PARALUTION` CPU and GPU implementations were performed over several solvers. Figure 7 presents such results for the following solvers:

- BiCGStab: Biconjugate gradient stabilized method [11]
- GMRES: Generalized minimal residual method [12]
- CG: Conjugate gradient method, requires symmetric matrix [13]
- CR: Conjugate residual method, requires symmetric matrix

Figure 7 shows that the covered GPU implementation provides the expected decrease in computation time. However, as the grid size increases, the gain obtained with GPU decreases. It is believed that the CPU results were slower than expected for small grid scales due to the `PARALUTION` implementations for CPU solvers. This library is designed for large scale systems which could explain the discrepancy observed. In the large scale region, a convergence towards an improvement factor around 3 is observed. Larger computations weren't available as the available hardware would run out of memory.

These results should be considered with respect to the following context: the CPU and GPU devices used present different hardware performances. A quantitative analysis would require further investigation in the performance characteristics of the device used.

One can observe how the results don't vary much over the different solvers. The influence of several preconditioners (Jacobi, Gauss-Seidel and SOR) was also investigated. The improvements observed were minor and are not reported in the present document.

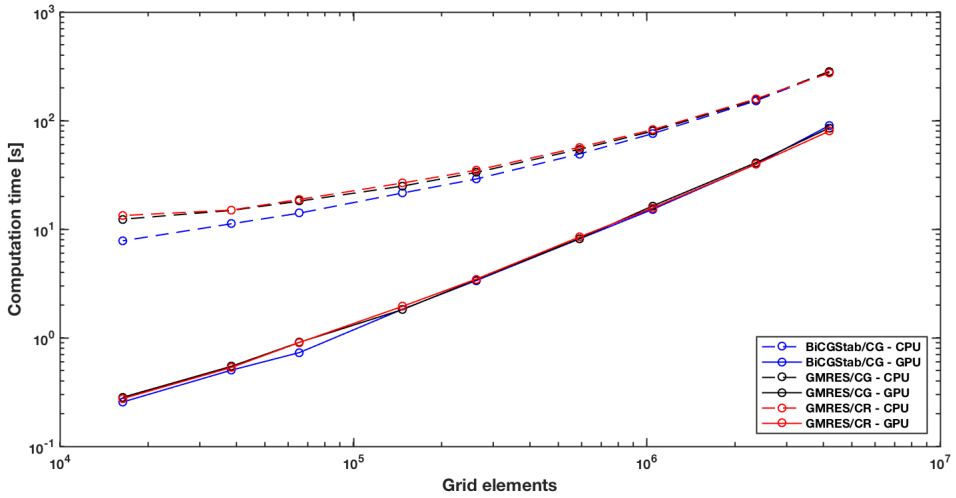

Figure 7: Comparisons between computation times for different solvers for the CPU and GPU `PARALUTION` implementations

4 Conclusions

This manuscript describes the methodology for successfully parallelizing the starter `mycavity` code using CPUs and GPUs. Call graph and code profile of the starter code are shown and it is demonstrated that the linear solver is most amenable to parallelization. A brief discussion on appropriate linear solvers for the problem is presented and the efficacy of direct solvers is first tested in the sequential case. Parallel solvers are then discussed, with dedicated sections on implementation and verification. Runtime reduction is observed upto 2048 cores and scaling studies performed on the multi-CPU version show favourable weak and strong scaling. The GPU version of the code is then discussed in detail and various iterative solvers are compared. A basic study on speedup yields a factor of 3 on porting the linear solver to the GPU. Further work would involve minimizing communication overheads and parallelizing flux computations.

5 Acknowledgements

This manuscript was written during as part of the ME469 course at Stanford University, run by Prof. Gianluca Iaccarino and Jared Dunnmon in Winter 2017. Their efforts towards the fruition of this project are gratefully acknowledged. This research effort used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy and special thanks to Prof. Matthias Ihme for making this possible. Also, the authors would like to thank Dr. Jack Poulson for his guidance and prompt responses, given that some of the rougher edges of the `Elemental` repository were being utilized.

A Supplementary information

The CPU implementations of the code are available on GitHub at https://github.com/gpavanb/mycavity/. The sequential version is on the `master` branch and the distributed version can be accessed in the `elemental` branch. The GPU version of the code is available in the `Paralution-GPU` branch. Instructions for compilation and running each version of the code are available in the online README file.

References

[1] H. L. Stone, Iterative solution of implicit approximations of multidimensional partial differential equations, SIAM Journal on Numerical Analysis 5 (3) (1968) 530–558.

[2] J. H. Ferziger, M. Peric, A. Leonard, Computational methods for fluid dynamics (1997).

[3] T. A. Davis, Direct methods for sparse linear systems, SIAM, 2006.

[4] X. S. Li, An overview of SuperLU: Algorithms, implementation, and user interface, ACM Transactions on Mathematical Software (TOMS) 31 (3) (2005) 302–325.

[5] C. Sanderson, et al., Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments.

[6] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, et al., LAPACK Users' guide.

[7] J. Poulson, B. Marker, R. A. Van de Geijn, J. R. Hammond, N. A. Romero, Elemental: A new framework for distributed memory dense matrix computations, ACM Transactions on Mathematical Software (TOMS) 39 (2) (2013) 13.

[8] J. Poulson, B. Engquist, S. Li, L. Ying, A parallel sweeping preconditioner for heterogeneous 3D Helmholtz equations, SIAM Journal on Scientific Computing 35 (3) (2013) C194–C212.

[9] J. R. Bunch, L. Kaufman, Some stable methods for calculating inertia and solving symmetric linear systems, Mathematics of computation (1977) 163–179.

[10] P. Labs, PARALUTION v1.1.0, http://www.paralution.com/ (2016).

[11] G. L. G. Sleijpen, H. A. van der Vorst, D. R. Fokkema, BiCGStab(l) and other hybrid Bi-CG methods, Numerical Algorithms 7 (1) (1994) 75–109.

[12] Y. Saad, M. H. Schultz, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM Journal on scientific and statistical computing 7 (3) (1986) 856–869.

[13] M. R. Hestenes, E. Stiefel, Methods of conjugate gradients for solving linear systems, Vol. 49, NBS, 1952.

[14] J. W. Demmel, J. R. Gilbert, X. S. Li, SuperLU User's guide, 1997.