Parallel Computing: Technology Trends
I. Foster et al. (Eds.)

doi:10.3233/APC200031

Parallel Smoothers in Multigrid Method for Heterogeneous CPU-GPU Environment

Neha IYER and Sashikumaar GANESAN [1]
Department of Computational and Data Sciences,
Indian Institute of Science, Bangalore 560012, India

Abstract. Modern-day supercomputers are equipped with sophisticated graphics processing units (GPUs) along with high-performance CPUs. Adapting existing algorithms specifically to GPU has resulted in under-utilization of CPU computing power. In this respect, we parallelize Jacobi and successive-over relaxation (SOR), which are used as smoother in multigrid method to maximize the combined utilization of both CPUs and GPUs. We study the performance of multigrid method in terms of total execution time by employing different hybrid parallel approaches, viz. accelerating the smoothing operation using only GPU across all multigrid levels, alternately switching between GPU and CPU based on the multigrid level and our proposed novel approach of using combination of GPU and CPU across all multigrid levels. Our experiments demonstrate a significant speedup using the hybrid parallel approaches, across different problem sizes and finite element types, as compared to the MPI only approach. However, the scalability challenge persists for the hybrid parallel multigrid smoothers.

Keywords. Parallel multigrid method, multi-GPU, multi-core, hybrid CPU-GPU

1. Introduction

Supercomputers today are equipped with multi-core CPU and multi-GPU to gain maximum performance. A single node of the top supercomputers supports up to 64 CPU cores with multiple GPUs. To utilize such massive computing power, there has been a significant effort to adapt existing algorithms onto the GPU architecture. In general, the compute extensive tasks are off-loaded to GPU while the CPU acts as a mediator performing data transfer, launching kernel call to the GPU or waiting idly in cases of blocking device API calls. Such practices have resulted in under-utilization of available CPU cores. The pressing need to utilize the combined computing capability of both GPUs and CPUs is even more relevant in case of small scale systems like personal computers that can support up to two GPU cards and up to eight CPU cores.

One of the centrepiece tasks that demand acceleration in the scientific computing community is solving a linear system of equations that generally arise from discretization of partial differential equations (PDEs) using a numerical method such as finite element. The multigrid method is considered to be the most efficient solver for such large

[1]Corresponding Author.
E-mail addresses: iyermohan@iisc.ac.in (N. Iyer), sashi@iisc.ac.in (S. Ganesan)

Table 1. Time taken by different operations at each level of multigrid V-cycle

Level	N	Pre-Smoothing Time (msec)	Post-Smoothing Time (msec)	Restriction (msec)	Prolongation (msec)
6	2,146,689	980.0	980.0	768.0	725.0
5	274,625	119.0	118.0	97.0	92.0
4	35,937	14.0	14.0	12.0	11.0
3	4,913	1.6	1.6	1.3	1.1
2	729	0.1	0.1	0.1	0.1
1	125	0.04			

sparse system of equations, with $O(N)$ computational complexity where N is the number of unknowns. Among the key operations of multigrid method, viz. smoothing, restriction and prolongation, smoothing is significantly time-consuming. Table 1 shows the time taken by different operations in a six-level geometric multigrid V-cycle. There is a notable difference in time taken by smoothing compared to other operations for fine mesh. In this respect, our main contribution is to improve the performance of geometric multigrid solver by developing hybrid parallel smoothers that concurrently utilizes all available computing resources in heterogeneous distributed systems. The hybrid parallel smoothers are implemented in our in-house finite element package ParMooN [1].

The rest of the paper is organized as follows: Section 2 discusses relevant work to accelerate the multigrid method. Section 3 gives a brief introduction to the geometric multigrid solver, the framework of ParMooN package and the model equation used for experiments. Section 4 describes the three different hybrid parallel approaches for smoothers. The experimental results and analysis are presented in Section 5 and Section 6 concludes the paper with key takeaways and future work.

2. Related Work

The problem of concurrent utilization of different computing resources has been previously studied in the literature. In [2], a parallel Jacobi iterative algorithm has been developed to exploit the computing capability of CPU along with the accelerators Xeon-Phi and GPU on a single node. In the case of multigrid method, existing approaches have adapted the key operations of smoothing and grid transfer to GPU architecture. The performance effect of combining the MPI only implementation of smoothers and grid transfer operators with either OpenMP or accelerators has been investigated in [3]. Another approach of mapping the fine level operations of geometric multigrid V-cycle to GPU and coarse level operations to CPU has been studied in [4]. The challenges of integrating existing MPI-based finite element package FEAST with GPU accelerated multigrid solvers has been presented in [5].

3. Background and Model Problem

3.1. Geometric Multigrid Method

Geometric multigrid (GMG) method is the most efficient iterative technique for solving a system of equations derived from a structured mesh. It operates on a hierarchy of meshes ranging from coarse to fine level l, where $l = 0,...,L$. The fine mesh is obtained by successively refining the coarse mesh L times uniformly. A typical multigrid γ-cycle is shown in Algorithm 1. For $\gamma = 1$ and $\gamma = 2$, the cycle becomes a V-cycle and W-cycle respectively. Classical iterative methods such as Jacobi or Successive-over relaxation (SOR) is used as a smoother due to their property of quick dampening of oscillatory modes. Multigrid method further leverages this property by recursively projecting the residual onto a coarser mesh where the smooth modes appear oscillatory. Few iterations of Jacobi or SOR work effectively on the coarse mesh and the computed correction is projected back to the fine mesh and used to update the original solution.

Algorithm 1 Multigrid γ-Cycle

```
1:  Procedure MG-CYCLE($l$)
2:  if $l == 0$ then
3:      Solve $A_l u_l = f_l$ exactly {coarsest level}
4:  else
5:      Apply pre-smoothing $\alpha$ times on $A_l u_l = f_l$ with an initial guess for $u_l$
6:      Restrict the residual to the next coarse level $f_{l-1} = R_l^{l-1}(f_l - A_l u_l)$, where $R$ is the
        restriction operator from $l$ to $(l-1)$ level
7:      Set $u_{l-1} = 0$
8:      for $j = 0$ to $\gamma_l$ do
9:          MG-CYCLE($l-1$)
10:     end for
11:     Prolongate $u_{l-1}$ to next fine level as $\tilde{u}_l = u_l + P_{l-1}^l u_{l-1}$, where $P$ is the prolongation
        operator from $(l-1)$ to $l$ level
12:     Apply post-smoothing $\alpha$ times on $A_l u_l = f_l$ with an initial guess as $\tilde{u}_l$
13: end if
```

3.2. Parallel Framework of ParMooN

In the parallel framework of ParMooN package [1], the input mesh is partitioned using METIS [6] software and the collection of cells is distributed across the MPI processes. Each process is allocated a sub-domain of cells on which it performs computations. Discretization of the domain leads to defining the degrees of freedom (DOFs) that constitute the unknowns. For a 3D mesh geometry, DOFs may be defined on vertices, edges, faces and in the interior of the cell based on the type of finite element used. Further, there are three types of finite element, viz. conforming, nonconforming and discontinuous type and we have considered conforming Q_1, Q_2 and nonconforming Q_1^{nc} type of finite elements as shown in Figure 1.

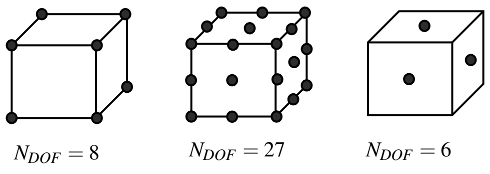

Figure 1. Conforming Q_1, Q_2 and nonconforming Q_1^{nc} finite element

Each MPI process classifies the DOFs into different types to facilitate communication of the solution with the neighbouring processes. The marking of DOFs in an individual process for a 2D domain is shown in Figure 2. The DOFs defined on the sub-domain boundaries are called Interface DOFs. These DOFs are shared by neighbouring MPI processes and the process that computes the solution at these DOFs mark the DOFs as Master DOF. All other processes sharing this DOF, mark it as Slave DOF. The DOFs that belong to the same process and are connected to Interface DOFs are called as Dependent DOFs. The Dependent DOF connected to a Slave DOF is called as Dependent1 DOF, otherwise it is called as Dependent2 DOF. The DOFs that belong to neighbouring processes but are connected to Interface DOFs are tagged as Halo DOFs. Further, Halo DOF connected to Master DOF is marked as Halo1, otherwise it is marked Halo2 DOF. The remaining DOFs owned by the process are defined as Independent DOFs.

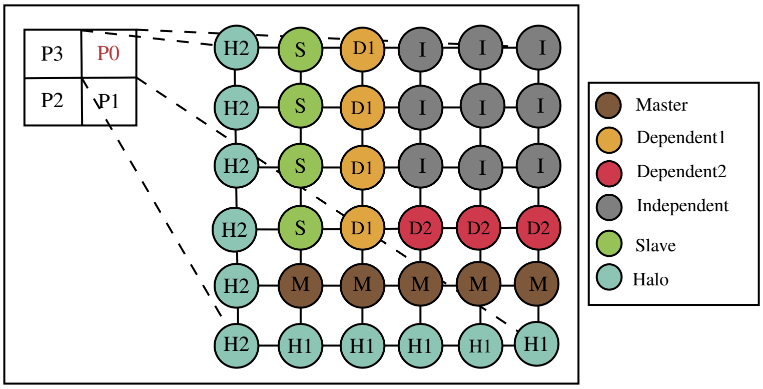

Figure 2. DOF Nomenclature using Q_1 finite element in a 2D domain for P0 process

The Master DOF in one process corresponds to Slave DOF in neighbouring processes. Similarly, Dependent1 and Dependent2 DOF correspond to Halo1 and Halo2 DOF. Hence, during each smoothing iteration, it is sufficient for each process to communicate the Master and Dependent1 DOFs with the neighbouring processes while the Dependent2 DOFs are communicated for every restriction and prolongation operations. The smoothing operation at each multigrid level including the coarsest level is performed using either Jacobi and SOR. The restriction and prolongation operators in ParMooN are as per [7] that proposed a general grid transfer operator between arbitrary finite element spaces.

3.3. Model Problem

We consider the steady-state Poisson equation with Dirichlet boundary condition on domain $\Omega \subseteq \mathbb{R}^3$ given by,

$$\begin{aligned} -\Delta u = f \quad &\text{in} \quad \Omega \\ u = 0 \quad &\text{on} \quad \partial\Omega. \end{aligned} \tag{1}$$

Here, u is the unknown scalar quantity and the source term f is chosen in such a way that the analytical solution $u = \sin(\pi x)\sin(\pi y)\sin(\pi z)$ satisfies equation (1). The equation is

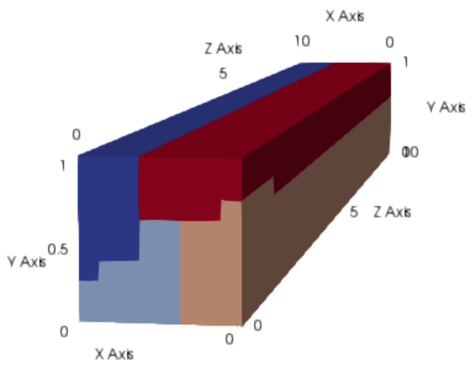

Figure 3. Cuboid domain partitioned among four MPI processes

solved in parallel by multiple MPI processes in ParMooN. The input domain is considered as a cuboid, shown in Figure 3 and is meshed using hexahedral cells. Equation (1) is discretized using standard Galerkin finite element method and subsequently the system of linear equations is solved up to a fixed precision using geometric multigrid V-cycles.

4. Implementation of Hybrid Parallel Smoother in Multigrid Method

4.1. DOF colouring

In SOR, each MPI process computes and communicates the DOFs in a pre-defined order based on the DOF type as shown in Algorithm 2. To perform the SOR iteration on GPU, the independent sets of DOFs that can be updated simultaneously must be identified. The colouring algorithm assigns a colour to each DOF such that no two connected DOFs of the same type have the same colour. A maximum of $O(d_m)$ colours will be used, where d_m is the maximum number of neighbours of the same type of DOF. The maximum neighbours of a DOF depend on the mesh structure and on the type of finite elements.

Algorithm 2 SOR Iteration at each MPI process

```
for $j = 0$ to $n_{smooth}$ do
    Compute Master DOF
    Communicate Master DOF
    Compute Dependent1 DOF
    Communicate Dependent1 DOF
    Compute Dependent2 DOF
    Compute Independent DOF
end for
```

Once the DOFs of all types are coloured, a CUDA kernel is launched to compute the DOFs that are assigned the same colour. Kernels are launched sequentially for each type of DOF as shown in Algorithm 2. DOF colouring step is not required in case of Jacobi iterations as the updated DOFs are computed using old DOF values. Hence, all DOFs can be updated in parallel.

4.2. Sparse Matrix-Vector Multiplication on GPU

The smoothing iterations in both Jacobi and SOR, involve repeated sparse matrix-vector multiplication (SpMV). The global stiffness matrix in ParMooN is stored in compressed sparse row (CSR) format. The CUDA kernels, CSR Scalar and CSR Vector proposed in [8] for performing SpMV in CSR format on GPU, are modified to perform the smoothing iterations. The CSR Scalar approach assigns a single CUDA thread whereas CSR Vector approach assigns a warp (32 threads) to perform a matrix row and vector multiplication. The performance benefits of both approaches are studied in our experiments.

4.3. Hybrid Parallel Approaches

We have designed three major approaches that decides whether the smoothing iteration is to be performed on GPU or CPU. The approaches are described as follows.

4.3.1. GPU only

In this approach, the smoothing iterations are performed on GPU for all types of DOFs and across all levels of multigrid. The iteration proceeds in the same manner as in Algorithm 2. For each type of DOF, a CUDA kernel is launched for each colour in case of SOR otherwise a single CUDA kernel is launched in case of Jacobi.

4.3.2. CPU-GPU non-overlapping

In this approach, the smoothing iterations are performed on GPU or CPU based on the level of the multigrid. A threshold multigrid level is empirically chosen such that the iterations on and above the chosen multigrid level are performed on GPU whereas the iterations below the threshold level are performed on CPU for all types of DOFs. As the system size is large on fine levels of multigrid, the ratio of number of DOFs to the total number of colours is high, thus allowing us to exploit fine-grained parallelism on GPU. At coarse levels of multigrid, the small system size makes CPU more suitable solver as it is better optimized for memory access.

4.3.3. CPU-GPU overlapping

In the case of SOR, each iteration is divided between CPU and GPU based on the type of DOF. The Independent DOFs constitute the major chunk of the total DOFs. Also, the Independent and Dependent2 DOFs need not be communicated during an iteration and hence these two types of DOFs are offloaded to GPU. The host process concurrently computes Master and Dependent1 DOFs and handles communication with other processes. The computation of boundary Independent DOFs require the updated Dependent1 DOF values and hence it is transferred from CPU to GPU and merged with GPU solution. Similarly, the Master DOFs are transferred to GPU and merged with GPU solution. The Independent and Dependent2 DOFs are transferred to CPU and merged with CPU solution at the end of the iteration. All merging operations are performed on the GPU and transferred back to CPU whenever required. Figure 4 shows a schematic representation of the various concurrent operations on CPU and GPU for a single SOR iteration.

In the case of Jacobi, the total DOFs are partitioned between GPU and CPU using an empirically chosen ratio of 4:1. At the end of each iteration, the GPU solution is merged with CPU solution.

The kernel calls to compute Independent DOFs are performed based on the ratio of the number of Independent DOFs to colours. If the ratio is greater than the empirically chosen value then a single kernel is launched per colour else the kernel calls for different colours are merged, thus trading off synchronous update to avoid kernel launch overhead.

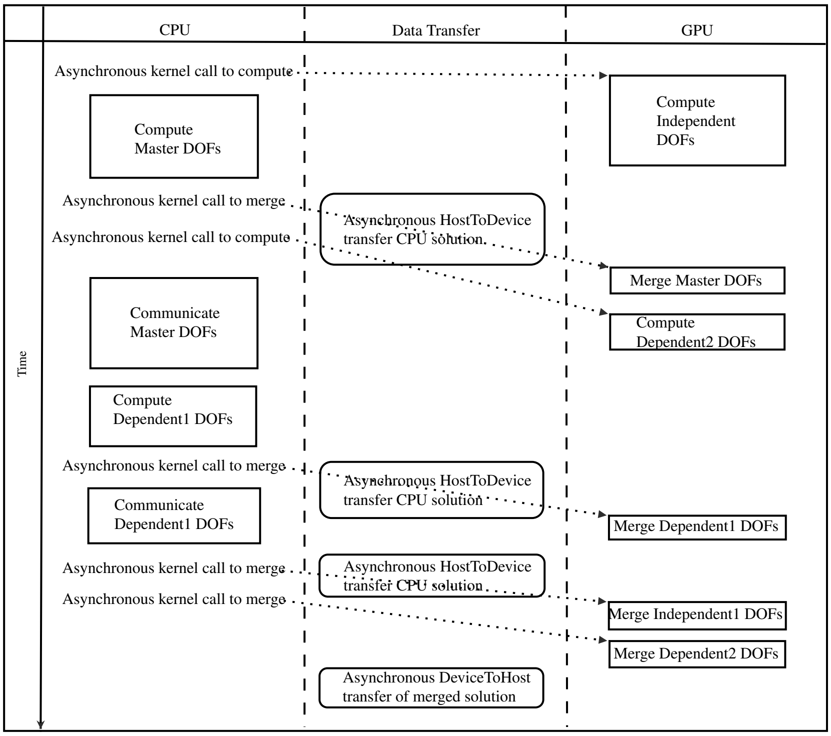
CPU
Data Transfer
GPU
Time
Asynchronous kernel call to compute
Compute Independent DOFs
Compute Master DOFs
Asynchronous kernel call to merge
Asynchronous kernel call to compute
Asynchronous HostToDevice transfer CPU solution
Merge Master DOFs
Communicate Master DOFs
Compute Dependent2 DOFs
Compute Dependent1 DOFs
Asynchronous kernel call to merge
Asynchronous HostToDevice transfer CPU solution
Communicate Dependent1 DOFs
Merge Dependent1 DOFs
Asynchronous kernel call to merge
Asynchronous HostToDevice transfer CPU solution
Asynchronous kernel call to merge
Merge Independent1 DOFs
Merge Dependent2 DOFs
Asynchronous DeviceToHost transfer of merged solution

Figure 4. Schematic representation of CPU-GPU overlapping algorithm for a single SOR iteration

4.4. CUDA optimizations

Different aspects of CUDA programming optimizations incorporated are as follows:

1. Minimum data transfer between CPU and GPU: The required data structures are transferred only once before the start of the smoothing step. During smoothing, only the solution array is transferred to merge the solution.
2. Maximum shared memory usage: The CUDA kernels use two separate shared memory arrays, one to perform parallel warp-wide reduction while computing the matrix row and vector dot product and the other to store the diagonal element of each row which is needed to update the DOF during the iteration.
3. Maximum CUDA occupancy: The threads per block value is set to maximize the occupancy value of each streaming multiprocessor (SM).
4. Implicit synchronization using warp-based operation: Since the CSR Vector approach uses warp-based approach to perform SpMV, no explicit synchronization is required within the CUDA kernel call.
5. Use of Multi-Process Service (MPS): We use MPS that allows the kernel and data transfer operations from multiple MPI processes to overlap on a single GPU.

5. Experimental Results

Experiments are performed to compare the strong scaling performance of the different hybrid parallel approaches with the existing MPI only approach of ParMooN for the following three variants:

1. Types of smoother: Jacobi and SOR
2. Size of problem: Small (100 K), medium (1000 K) and large (10000 K)
3. Types of finite element: conforming Q_1, Q_2 and nonconforming Q_1^{nc}

The variable multigrid parameters used for different experiments are listed in Table 2. The experiments are executed on CRAY XC40 machine at SERC, Indian Institute of Science, Bangalore [9]. A single node in the cluster has an Intel IvyBridge 2.4 GHz based single CPU socket with 12 cores along with an NVIDIA Tesla K40 GPU card with 2880 cores and 12GB device memory. The small and medium size problem experiments are performed using four nodes with up to eight CPU cores per node and large size problems are performed using up to eight nodes with eight CPU cores per node.

Table 2. Multigrid Solver Parameters

Smoother	**FE type**	**Levels**	**N**	n_{pre}	n_{post}	n_{coarse}	$\omega_{smoother}$
Jacobi	Q_2	4	2000 K	5	5	10	0.67
	Q_1^{nc}	5	6000 K				
	Q_1	4	2000 K				
SOR	Q_2	4	2000 K	5	5	10	1.00
	Q_1^{nc}	5	6000 K				
	Q_1	6	17000 K				
		5	2000 K				
		4	300 K				

5.1. Scaling performance of hybrid parallel smoothers using different finite elements

Figure 5 shows the total time taken by the geometric multigrid solver using different hybrid parallel approaches for the smoother. We use CSR Vector approach for smoothing iteration on GPU. The total time includes solving as well as communication time between neighbouring processes of the multigrid solver. At low scale, the hybrid parallel approaches applied to smoothers have reduced the solving time significantly compared to MPI only approach and that too, across different finite elements. The average reduction in solving time across different finite elements for two MPI processes is 39% and 77% using Jacobi and SOR, respectively. The hybrid parallel approaches however, are not able to perform consistently at higher scales. The poor scaling performance of GPU approaches may be attributed to the decrease in problem size per process causing reduced parallelism and significant increase in data transfer and kernel launching overheads.

Among the three hybrid parallel approaches, the performance of GPU only and CPU-GPU non-overlapping are quite comparable. The GPU only performs better at low scale whereas the performance of CPU-GPU non-overlapping gets better at higher scales because of the involvement of CPU at coarse levels of multigrid. CPU-GPU overlapping approach performs slightly poorer at low scales since the CPU workload is higher and hence takes more time to complete the computation as compared to GPU. However, the performance matches with other approaches as the scale increases.

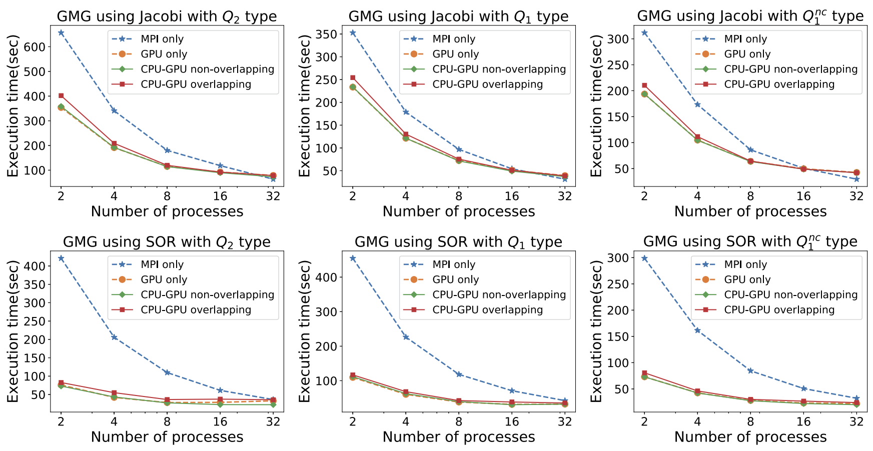

Figure 5. Scaling performance of GMG solver using Jacobi and SOR smoothers for conforming Q_1, Q_2 and nonconforming Q_1^{nc} finite elements for medium size problems (1000 K)

5.2. *Performance of hybrid parallel smoothers across different problem sizes*

The performance trend of the hybrid parallel approaches is tested across different problem sizes. Table 3 shows the speedup obtained across three different problem sizes of 300 K(small), 2000 K(medium) and 17000 K(large) with SOR smoother and Q_1 finite element. For small problem size, the performance slowly degrades with an increasing number of processes using hybrid parallel smoother. The speedup achieved using each of the approaches increases on increasing the problem size. The speedup trend for GPU only and CPU-GPU non-overlapping is almost comparable. The CPU-GPU non-overlapping results in better speedup compared to GPU only, as the scale of processes increases.

Table 3. Speedup of hybrid parallel approaches to MPI only approach across different problem sizes

N	GPU only				CPU-GPU overlapping				CPU-GPU non-overlapping			
	Number of processes				Number of processes				Number of processes			
	4	8	16	32	4	8	16	32	4	8	16	32
300 K	1.97	1.26	0.72	0.31	1.20	1.15	0.71	0.31	1.38	1.47	0.93	0.42
2000 K	3.73	3.09	2.25	1.31	3.17	2.62	1.80	1.16	3.61	2.99	2.32	1.32
17000 K	3.89	4.17	3.77	3.05	3.90	3.69	2.96	2.41	4.30	4.19	3.80	3.12

5.3. *Performance gain using CSR Vector*

The CSR Scalar and CSR Vector approaches are tested for conforming Q_1, Q_2 and nonconforming Q_1^{nc} finite elements using SOR smoother. Figure 6 shows the speedup achieved using CSR Vector compared to CSR Scalar using GPU only approach.

CSR Vector performs better across all the considered finite elements. The higher-order finite element particularly Q_2 type benefit more ($\sim$ 3 times) using the CSR Vector approach as there are more number of non-zeroes in each matrix row thus exploiting fine-grained parallelism. This reduction in solving time can be leveraged to compensate for higher communication time observed in general for higher-order finite elements.

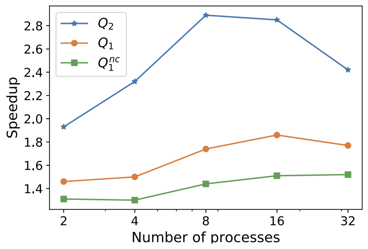

Figure 6. CSR Vector and CSR Scalar comparison for different finite elements

6. Conclusions

We have implemented and analyzed three different hybrid parallel approaches for multigrid smoother. The GPU only and CPU-GPU non-overlapping approaches give better speedup in certain scales compared to MPI only, but fail to utilize all computing resources simultaneously in a heterogeneous distributed system. The proposed novel CPU-GPU overlapping approach overcomes this drawback and performs comparably to both GPU only and CPU-GPU non-overlapping, provided the ratio of computing speed to workload is balanced between CPU and GPU. Individually, the studied approaches shows poor scalability. The GPU only gives good performance benefits at fine mesh having large problem size. The CPU-GPU overlapping works well on the intermediate mesh where there is better load balancing. On the coarse mesh, MPI only works best with small problem size. This leads us to explore the use of a combination of different approaches to further optimize the performance of the multigrid method. In future, we have planned to improve the scaling behaviour by deriving a heuristic to switch between the hybrid parallel approaches based on the mesh size and the number of MPI processes.

References

[1] S. Ganesan, V. John, G. Matthies, R. Meesala, A. Shamim, and U. Wilbrandt, "An object oriented parallel finite element scheme for computations of PDEs: Design and implementation," in *2016 IEEE 23rd International Conference on High Performance Computing Workshops (HiPCW)*, pp. 106–115, IEEE, 2016.

[2] S. Contassot-Vivier and S. Vialle, "Algorithmic scheme for hybrid computing with CPU, Xeon-Phi/MIC and GPU devices on a single machine," *Parallel Computing: On the Road to Exascale*, vol. 27, p. 25, 2016.

[3] M. Wlotzka and V. Heuveline, "Energy-efficient multigrid smoothers and grid transfer operators on multicore and GPU clusters," *Journal of Parallel and Distributed Computing*, vol. 100, pp. 181–192, 2017.

[4] N. Sakharnykh, "High-Performance Geometric Multi-Grid with GPU Acceleration," Feb. 2016. `https://devblogs.nvidia.com/high-performance-geometric-multi-grid-gpu-acceleration/`.

[5] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. S. McCormick, H. Wobker, C. Becker, and S. Turek, "Using gpus to improve multigrid solver performance on a cluster," *International Journal of Computational Science and Engineering*, vol. 4, no. 1, p. 36, 2008.

[6] G. Karypis and V. Kumar, "METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0," tech. rep., 1995.

[7] F. Schieweck, "A general transfer operator for arbitrary finite element spaces," 2000.

[8] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," tech. rep., Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[9] "SahasraT." `http://www.serc.iisc.ac.in/cray-xc40-named-as-sahasrat/`.