1-1-2013

Multi-Level Parallelism for Incompressible Flow Computations on GPU Clusters

Dana A. Jacobsen
Boise State University

Inanc Senocak
Boise State University

DOI: 10.1016/j.parco.2012.10.002

Multi-level Parallelism for Incompressible Flow Computations on GPU Clusters

Dana A. Jacobsen[a], Inanc Senocak[b,*]

[a] *Department of Computer Science*
[b] *Department of Mechanical and Biomedical Engineering*
[c] *Boise State University, Boise, ID 83725*

Abstract

We investigate multi-level parallelism on GPU clusters with MPI-CUDA and hybrid MPI-OpenMP-CUDA parallel implementations, in which all computations are done on the GPU using CUDA. We explore efficiency and scalability of incompressible flow computations using up to 256 GPUs on a problem with approximately 17.2 billion cells. Our work addresses some of the unique issues faced when merging fine-grain parallelism on the GPU using CUDA with coarse-grain parallelism that use either MPI or MPI-OpenMP for communications. We present three different strategies to overlap computations with communications, and systematically assess their impact on parallel performance on two different GPU clusters. Our results for strong and weak scaling analysis of incompressible flow computations demonstrate that GPU clusters offer significant benefits for large data sets, and a dual-level MPI-CUDA implementation with maximum overlapping of computation and communication provides substantial benefits in performance. We also find that our tri-level MPI-OpenMP-CUDA parallel implementation does not offer a significant advantage in performance over the dual-level implementation on GPU clusters with two GPUs per node, but on clusters with higher GPU counts per node or with different domain decomposition strategies a tri-level implementation may exhibit higher efficiency than a dual-level implementation and needs to be investigated further.

Keywords: GPU, hybrid MPI-OpenMP-CUDA, fluid dynamics

*Corresponding author
Email addresses: dana@acm.org (Dana A. Jacobsen), senocak@boisestate.edu (Inanc Senocak)

1. Introduction

Many applications in advanced modeling and simulation require more resources than a single computing unit can provide, whether in the problem size or the required performance. Graphics processing units (GPUs) have enjoyed rapid adoption within the high-performance computing (HPC) community because GPUs enable high levels of fine-grain data parallelism. The latest GPU programming interfaces such as NVIDIA's Compute Unified Device Architecture (CUDA) [1], and more recently Open Computing Language (OpenCL) [2] provide the programmer a flexible model while exposing enough of the hardware for optimization.

Current high-end GPUs can achieve high floating point throughputs by combining highly parallel processing (200-800 scalar processing units per GPU), high memory bandwidth and efficient thread scheduling. GPU clusters, where fast network connected compute-nodes are augmented with latest GPUs, [3] are now being used to solve challenging problems from various domains. Examples include the 384 GPU Lincoln Tesla cluster operated by the National Center for Supercomputing Applications (NCSA) at University of Illinois at Urbana Champaign [4] and the 512 GPU Longhorn cluster at the Texas Advanced Computing Center (TACC). Latest supercomputers too allow large numbers of GPUs to be used to solve single problems. Examples include the 7168 GPU Tianhe-1A [5, 6] and the 4640 GPU Dawning Nebulae [7] supercomputers. These new systems are designed for high performance as well as high power efficiency, which is a crucial factor in future exascale computing [8].

2. Related Works

GPU computing has evolved from hardware rendering pipelines that were not amenable to non-rendering tasks, to the modern General Purpose Graphics Processing Unit (GPGPU) paradigm. Owens et al. [9] survey the early history as well as the state of GPGPU computing up to 2007. The use of GPUs for Euler solvers and incompressible Navier-Stokes solvers has also been well documented [10–17].

Modern motherboards can accommodate multiple GPUs in a single workstation with several TeraFLOPS of peak performance, but GPU programming models have to be interleaved with MPI, OpenMP or Pthreads to make

use of all the GPUs in computations. In the multi-GPU computing front, Thibault and Senocak [15, 16] developed a single-node multi-GPU 3D incompressible Navier-Stokes solver with a Pthreads-CUDA implementation. The GPU kernels from their study forms the internals of the present cluster implementation. Thibault and Senocak demonstrated a speedup of 21× for two Tesla C870 GPUs compared to a single core of an Intel Core 2 E8400 3.0 GHz processor, 53× for two GPUs compared to an AMD Opteron 8216 2.4 GHz processor, and 100× for four GPUs compared to the same AMD Opteron processor. Four GPUs were able to sustain 3× speedup compared to a single GPU on a large problem size. The multi-GPU implementation of Thibault and Senocak does not overlap computation with GPU data exchanges. Therefore, three overlapping strategies are systematically introduced and evaluated in the present study.

Micikevicius [18] describes both single and multi GPU CUDA implementations of a 3D 8^{th}-order finite difference wave equation computation. The wave equation code is composed of a single kernel with one stencil operation, unlike CFD computations which consist of multiple inter-related kernels. MPI was used for process communication in multi-GPU computing. Micikevicius uses a two stage computation where the cells to be exchanged are computed first, then the inner cells are computed in parallel with asynchronous memory copy operations and MPI exchanges. With efficient overlapping of computations and copy operations, Micikevicius achieves very good scaling on 4 GPUs running on two Infiniband connected nodes with two Tesla 10-series GPUs each, when using a large enough dataset.

Göddeke et al. [12] explored course and fine grain parallelism in a finite element model for fluids or solid mechanics computations on a GPU cluster. Göddeke et al. [19] described the application of their approach to a large-scale solver toolkit. The Navier-Stokes simulations in particular exhibited limited performance due to memory bandwidth and latency issues. Optimizations were also found to be more complicated than simpler models such as the ones they previously considered. While the small cluster speedup of a single kernel is good, unfortunately acceleration of the entire model is only a modest factor of two. Their model uses a nonuniform grid and multigrid solvers within a finite element framework for relatively low Reynolds numbers.

Phillips et al. [20] describe many of the challenges that arise when implementing scientific computations on a GPU cluster, including the host/device memory traffic and overlapping execution with computation. A performance visualization tool was used to verify overlapping of CPU, GPU, and commu-

nication on an Infiniband connected 64 GPU cluster. Scalability is noticeably worse for the GPU accelerated application than the CPU application as the impact of the GPU acceleration is quickly dominated by the communication time. However, the speedup is still notable. Phillips et al. [21] describe a 2D Euler Equation solver running on an 8 node cluster with 32 GPUs. The decomposition is 1D, but GPU kernels are used to gather/scatter from linear memory to non-contiguous memory on the device.

While MPI is the API typically used for network communication between compute nodes, it presents a distributed memory model which can potentially make it less efficient for processes running on the same shared-memory compute node [22, 23]. For this reason, hybrid programming models combining MPI and a threading model such as OpenMP or Pthreads have been proposed with the premise that message passing overhead can be reduced, increasing scalability. With two to four GPUs per compute node, a hybrid MPI-OpenMP-CUDA method warrants further investigation and is studied in this paper along with an MPI-CUDA method to develop a multi-level parallel incompressible flow solver for GPU clusters.

Cappello, Olivier, and Etiemble [24–26] were among the first to present the hybrid programming model of using MPI in conjunction with a threading model such as OpenMP. They demonstrated that it is sometimes possible to increase efficiency on some codes by using a mixture of shared memory and message passing models. A number of other papers followed with the same conclusions [27–34]. Many of these papers also point out a number of cases where the applications or computing systems are a poor fit to the hybrid model, and in some cases performance decreases. Lusk and Chan [35] describes using OpenMP and MPI for hybrid programming on three cluster environments, including the effect the different models have on communication with the NAS benchmarks. They claim combination of MPI and OpenMP parallel programming is well fitted to modern scalable high performance systems.

Hager, Jost, and Rabenseifner [36] give a recent perspective on the state of the art techniques in hybrid MPI-OpenMP programming. Particular attention is given to mapping the model to domain decomposition as well as overlapping methods. Results with hybrid models of the BT-MZ benchmark (part of the NAS Parallel Benchmark suite) on a Cray XT5 using a hybrid approach showed similar performance at 64 and fewer cores, but greatly improved results for 128, 256, and 512 cores, where a good combination of OpenMP fine-grain parallelism combined with MPI coarse-grin parallelism

can be found that matches well with the hardware. These examples also take advantage of the loop scheduling features in OpenMP. Advantages in fine grain parallelism like this will not be able to be taken advantage of in a model where OpenMP is only used for coarse-grain data transfer and synchronization.

Balaji et al. [37] discuss issues arising from using MPI on petascale machines with close to a million processors. A number of irregular MPI collective operations are considered to be nonscalable when applied to a very large number of processes. The tested MPI implementations also allocate some memory which is proportional to the number of processes, limiting scalability. These as well as other limitations lead Balaji et al. to suggest a hybrid threading / MPI model as one way to mitigate the issue. However, we think, in the case of a typical GPU system the situation is not as bad. Because the CUDA model for fine-grain parallelism manages 256 to 512 processing elements within a single process, and this number will likely increase with future GPUs. Hence a one million processing element GPU cluster using just MPI-CUDA may have fewer than 4000 MPI processes. This suggests that clusters enhanced with GPUs look well suited for petascale and emerging exascale architectures. Therefore, compute-intensive applications need to be evaluated for parallel efficiency and performance on large GPU clusters. Our study is one of few that critically evaluates multi-level parallelism of incompressible flow computations on GPU clusters.

Nakajima [38] describes a three-level hybrid method using MPI, OpenMP, and vectorization. This approach uses MPI for inter-node communication, OpenMP for intra-node communication, and parallelism within the node via the vector processor. It closely matches the rationale behind our hybrid MPI-OpenMP-CUDA approach for a GPU cluster implementation. Nakajima's weak scaling measurements showed worse results for 64 and fewer SMP nodes, but improved with 96 or more. GPU clusters with 128 or more compute-nodes (256 or more GPUs) are rare at this time, but trends indicate these machines will become far more common in the high performance computing field [6–8].

While these articles show some potential benefits for using the hybrid model on CPU clusters, a question is whether the same benefits will accrue to a tri-level CUDA-OpenMP-MPI model, and whether the benefits will outweigh the added software complexity. With high levels of data parallelism on the GPU, separate memory for each GPU, low device counts per node, and currently small node counts, the GPU cluster model has numer-

ous differences from dense-core CPU clusters. In this paper we investigate several methods of distributing computation using a dual (MPI-CUDA) and tri-level (MPI-OpenMP-CUDA) parallel programming approaches along with different strategies to overlap computation and communication on GPU clusters. We adopt MPI for coarse-grain inter-node communication, OpenMP for medium-grain intra-node communication in the tri-level approach, and CUDA for fine-grain parallelism within the GPUs. In all of our implementations, computations are entirely done on the GPU using CUDA. We use a 3D incompressible flow Navier-Stokes solver to systematically assess scalability and performance of multi-level parallelism on large GPU clusters.

3. Governing Equations and Numerical Approach

Navier-Stokes equations for buoyancy driven incompressible fluid flows can be written as follows:

$$\nabla \cdot \mathbf{u} = 0, \tag{1}$$

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} = -\frac{1}{\rho}\nabla P + \nu \nabla^2 \mathbf{u} + \mathbf{f}, \tag{2}$$

where $\mathbf{u}$ is the velocity vector, P is the pressure, ρ is the density, ν is the kinematic viscosity, and $\mathbf{f}$ is the body force. The Boussinesq approximation, which applies to incompressible flows with small temperature variations, is used to model the buoyancy effects in the momentum equations [39]:

$$\mathbf{f} = \mathbf{g} \cdot (1 - \beta(T - T_\infty)), \tag{3}$$

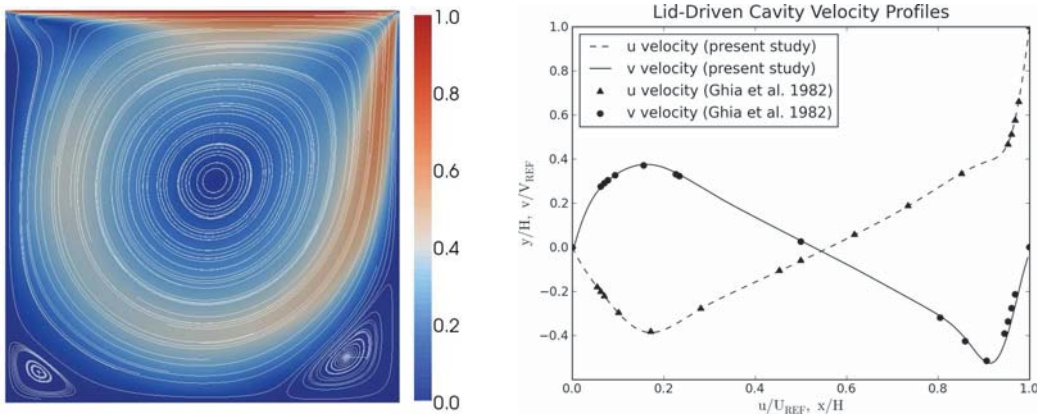
where $\mathbf{g}$ is the gravity vector, β is the thermal expansion coefficient, T is the calculated temperature at the location, and T_∞ is the steady state temperature.

The temperature equation can be written as [40, 41]

$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{u}T) = \alpha \nabla^2 T + \Phi, \tag{4}$$

where α is the thermal diffusivity and Φ is the heat source.

The buoyancy-driven incompressible form of the Navier-Stokes equations (Eqs. 1–4) do not have an explicit equation for pressure. Therefore, we use the projection algorithm of Chorin [42], where the velocity field is first predicted using the momentum equations without the pressure gradient term.

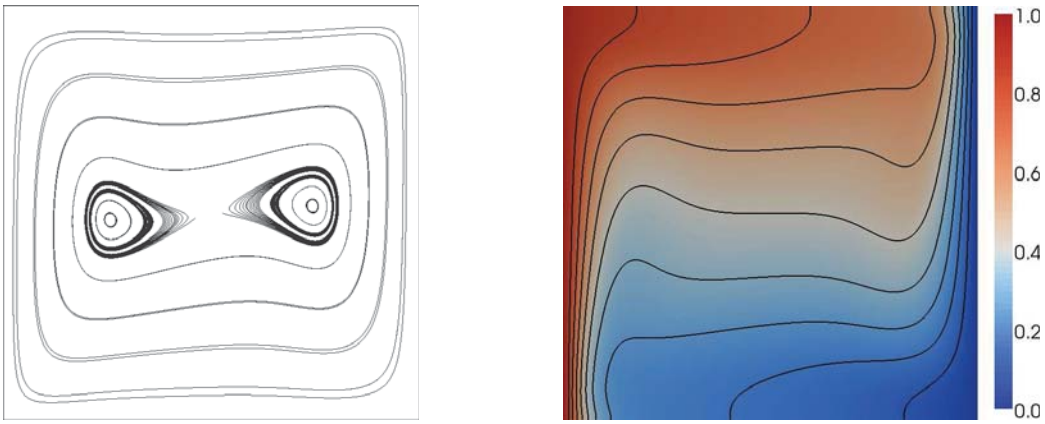


a) b)

Figure 1. Lid-driven cavity simulation with Re = 1000 on a $256 \times 32 \times 256$ grid. 3D computations were used and a 2D center slice is shown. a) Velocity streamlines and velocity magnitude distribution. b) Comparison to the benchmark data from Ghia et al. [44].

The resulting predicted velocity field does not satisfy the divergence free condition. The divergence free condition is then enforced on the velocity field at time $t + 1$, to derive a pressure Poisson equation from the momentum equations given in Eq. (2). We solve the discretized versions of the resulting equations on a uniform Cartesian staggered grid with second order central difference scheme for spatial derivatives and a second order accurate Adams-Bashforth scheme for time derivatives. The pressure Poisson equation can be solved using either a fixed iteration Jacobi solver or a parallel geometric multigrid solver [43]. Both solvers are available in our code. We do not activate the geometric multigrid solver in certain computations where we investigate dual- and tri-level parallelism, because the amalgamated parallel implementation of the multigrid method complicates the detailed analysis of scaling and breakdown of communication timings due to the inherent algorithmic complexity in the method.

Validation on a number of test cases including the well-known lid-driven cavity and natural convection in heated cavity problems [44, 45] were used to compare the overall solutions to known results. Figure 1 presents the results of a lid-driven cavity simulation with a Reynolds number 1000 on a $256 \times 32 \times 256$ grid. Figure 1a shows the velocity magnitude distribution



a) b)

Figure 2. Natural convection in a cavity using a $128 \times 16 \times 128$ grid and Prandtl number 7, with a 2D center slice shown. a) Streamlines for Rayleigh number 200,000. b) Isotherms and temperature distribution for Rayleigh number 200,000.

and streamlines at mid-plane. As expected, the computations capture the two corner vortices at steady-state. In Fig. (1b), the horizontal and vertical components of the velocity along the centerlines ar e compared to the benchmark data of Ghia et al. [44]. The results agree well with the benchmark data. The numerical results for the tri-level and dual-level parallel versions do not differ.

We simulate the natural convection in a heated cavity problem to test our buoyancy-driven incompressible flow computations on a $128 \times 16 \times 128$ grid. Figure 2 presents the natural convection patterns and isotherms for Rayleigh (Ra) numbers of 200,000 and a Prandtl (Pr) number of 7.0. Lateral walls have constant temperature boundary conditions with one of the walls having a higher temperature than the wall on the opposite side. Top and bottom walls are insulated. Fluid inside the cavity is heated on the hot lateral wall and rises due to buoyancy effects, whereas on the cold wall it cools down and sinks, creating a circular convection pattern inside the cavity. Although not shown in the present paper, our results agree well with similar results presented in Griebel et al. [40]. A direct comparison is available in Jacobsen [17]. Figure 3 presents a comparison of the horizontal centerline temperatures for a heated cavity with Ra=100,000 and Pr=7.0 along with reference data

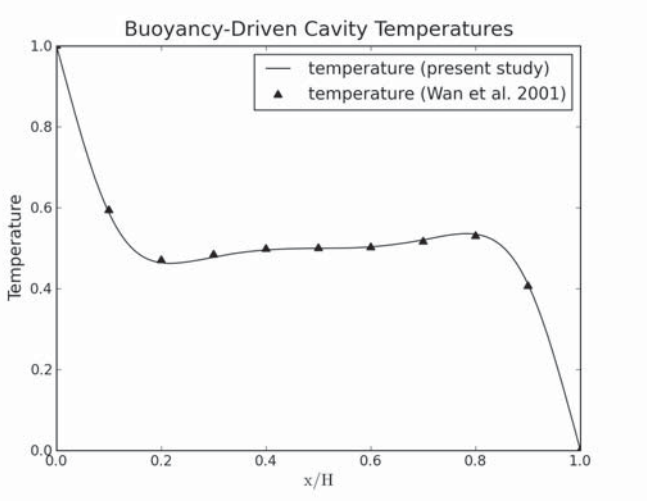


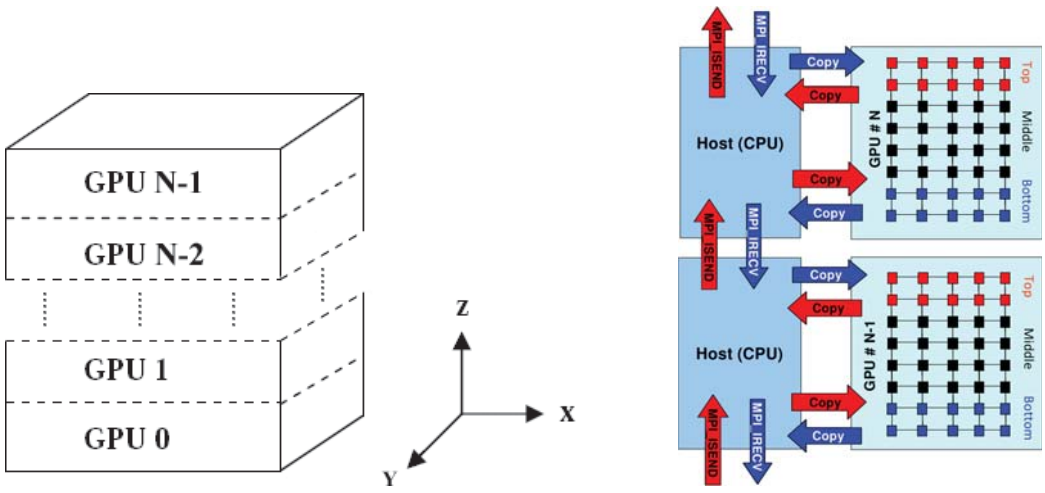
Figure 3. Centerline temperature for natural convection in a cavity with Prandtl number 7 and Rayleigh number 100,000, using a $256 \times 16 \times 256$ grid with a 2D center slice used. Comparison is shown to data from Wan et al. [45].

from Wan et al. [45]. Our results are in very good agreement.

Aside from these benchmark cases, our CFD solver can compute flow around embedded obstacles such as urban areas and complex terrain can be found in [17, 46, 47]

4. Multi-level Parallelism

Multiple programming APIs along with a domain decomposition strategy for data-parallelism is required to achieve high throughput and scalable results from a CFD model on a multi-GPU platform. For problems that are small enough to run on a single GPU, overhead time is minimized as no GPU/host communication is performed during the computation, and all optimizations are done within the GPU code. When more than one GPU is used, cells at the edges of each GPU's computational space must be communicated to the GPUs that share the domain boundary so they have the current data necessary for their computations. Data transfers across the neighboring GPUs inject additional latency into the implementation which can restrict scalability if not properly handled. For these reasons we investigate multi-level parallelism on GPU clusters with different implementations



a) b)

Figure 4. The domain decomposition. a) The decomposition of the full computational domain to the individual GPUs. b) An overview of the communication, GPU memory transfers, and the intra-GPU 1D decomposition used for overlapping.

to improve the performance and scalability of our Navier-Stokes solver.

4.1. Domain Decomposition

A 3D Cartesian volume is decomposed into 1D slices. These slices are then partitioned among the GPUs on the cluster to form a 1D domain decomposition. The 1D decomposition is shown in Figure 4a. After each GPU completes its computation, the edge cells ("ghost cells") must be exchanged with neighboring GPUs. Efficiently performing this exchange process is crucial to cluster scalability as we demonstrate in section 5.

While a 1D decomposition leads to more data being transferred as the number of GPUs increases, there are advantages to the method when using CUDA. In parallel CPU implementations, host memory access can be performed on non-contiguous segments with a relatively small performance loss. The `MPI_CART` routines supplied by MPI allow efficient management of virtual topologies, making the use of 2D and 3D decompositions easy and efficient. In contrast, the CUDA API only provides a way to transfer linear segments of memory between the host and the GPU. Hence, 2D or 3D decompositions for GPU implementations must either use nonstandard device

memory layouts which may result in poor GPU performance, or run separate kernels to perform gather/scatter operations into a linear buffer suitable for the `cudaMemcpy()` routine. These routines add significant time and hinder overlapping methods. For these reasons, the 1D decomposition was deemed best for moderate size clusters such as the ones used in this study.

To accommodate overlapping, a further 1D decomposition is applied within each GPU. Figure 4b indicates how the 1D slices within each GPU are split into a top, bottom, and middle section. When overlapping communication and computation, the GPU executes each separately such that the memory transfers and MPI communication can happen simultaneously with the computation of the middle portion.

4.2. Dual-Level MPI-CUDA Implementations

The work by Thibault and Senocak [15, 16] showed how an incompressible Navier-Stokes solver written for a single GPU can be extended to multiple GPUs by interleaving CUDA with Pthreads. The full 3D domain is decomposed across threads in one dimension, splitting on the Z axis. The resulting partitions are then solved using one GPU per thread. No effort was made to hide latencies arising from GPU data transfers or Pthreads synchronization. To solve the restrictions of the shared memory model of Thibault and Senocak, we adopt MPI as the mechanism for communication between GPUs, and introduce three strategies to overlap computations on the GPU with data copying to and from the GPU and MPI communication across the network.

In our present implementation, a single MPI process is started per GPU, and each process is responsible for managing its GPU and exchanging data with its neighbor processes. Since we must ensure that each process is assigned a unique GPU identifier, an initial mapping of hosts to GPUs is performed. A master process gathers all the host names, assigns GPU identifiers to each host such that no process on the same host has the same identifier, and scatters the result back. At this point the `cudaSetDevice()` call is made on each process to map one of the GPUs to the process which assures that no other process on the same node will map to the same GPU. All ghost cell exchanges are done via `MPI Isend` and `MPI_Irecv`. Overlap of computations with inter-node and intra-node data exchanges is accomplished to better utilize the cluster resources. All three of the implementations have much in common, with differences in the way data exchanges are implemented. It is shown in section 5 that implementation details in the data exchanges have a large impact on performance.

```
for (t=0; t < time_steps; t++)
{
    adjust_timestep();

    for (stage = 0; stage < num_timestep_stages; stage++) {
        temperature <<<grid,block>>> (u,v,w,phiold,phi,phinew);
        ROTATE_POINTERS(phi,phinew);
        temperature_bc <<<grid,block>>> (phi);
        EXCHANGE(phi);

        turbulence <<<grid,block>>> (u,v,w,nu);
        turbulence_bc <<<grid,block>>> (nu);
        EXCHANGE(nu);

        momentum <<<grid,block>>> (phi,uold,u,unew,vold,v,vnew,wold,w,wnew);
        momentum_bc <<<grid,block>>> (unew,vnew,wnew);
        EXCHANGE(unew,vnew,wnew);
    }

    divergence <<<grid,block>>>(unew,vnew,wnew,div);

    // Iterative or multigrid solution
    pressure_solve(div,p,pnew);

    correction <<<grid,block>>> (unew,vnew,wnew,p);
    momentum_bc <<<grid,block>>> (unew,vnew,wnew);
    EXCHANGE(unew,vnew,wnew);
    ROTATE_POINTERS(u,unew);  ROTATE_POINTERS(v,vnew);  ROTATE_POINTERS(w,wnew);
}
```

Listing 1. Host code for the projection algorithm to solve buoyancy driven incompressible flow equations on multi-GPU platforms. The `EXCHANGE` step updates the ghost cells for each GPU with the contents of the data from the neighboring GPU.

```
// PART 1:  Interleave non-blocking MPI calls with device
//          to host memory transfers of the edge layers.

// Communication to south
MPI_Irecv(new ghost layer from north)
cudaMemcpy(south edge layer from device to host)
MPI_Isend(south edge layer to south)
// Communication to north
MPI_Irecv(new ghost layer from south)
cudaMemcpy(north edge layer from device to host)
MPI_Isend(north edge layer to north)

// ... other exchanges may be started here, before finishing in order

// PART 2:  Once MPI indicates the ghost layers have been received,
//          perform the host to device memory transfers.

MPI_Wait(new ghost layer from north)
cudaMemcpy(new north ghost layer from host to device)
MPI_Wait(new ghost layer from south)
cudaMemcpy(new south ghost layer from host to device)
MPI_Waitall(south and north sends, allowing buffers to be reused)
```

Listing 2. An `EXCHANGE` operation overlaps GPU memory copy operations with asynchronous MPI calls for communication.

The projection algorithm is composed of distinct steps in the solution of the fluid flow equations. Listing 1 shows an outline of the basic implementation using CUDA kernels to perform each step. The steps marked as `EXCHANGE` are where ghost cells for each GPU are filled in with the calculated contents of their neighboring GPUs. The most basic exchange method is to call `cudaMemcpy()` to copy the edge data to host memory, MPI exchange using `MPI Send` and `MPI_Recv`, and finally `cudaMemcpy()` to copy the received edge data to device memory. This is straightforward, but all calls are blocking which greatly hinders performance. Therefore, we have not pursued this basic implementation in the present study.

4.2.1. Non-blocking MPI with No Overlapping of Computation

The first implementation uses non-blocking MPI calls [50] to offer a substantial benefit over the blocking approach, which we do not pursue. Our first implementation does not overlap computation although it tries to over-

lap memory copy operations. The basic `EXCHANGE` operation is shown in Listing 2. In this approach, none of the device/host memory operations nor any MPI communication happens until the computation of the entire domain has completed. The MPI communication is able to overlap with the CUDA memory operations. When multiple arrays need to be exchanged, such as the three momentum components, the components may be interleaved such that the MPI send and receive for one edge of the first component is in progress while the memory copy operations for the later component are proceeding. This is done by starting part 1 for each component in succession, then part 2 for each component.

4.2.2. Overlapping Computation with MPI Communications

The second implementation for exchanges aims to overlap the CUDA computation with the CUDA memory copy operations and the MPI communication. We split the CUDA kernels into three calls such that the edges can be done separately from the middle. This has a very large impact on the cluster performance as long as the domain is large enough to give each GPU enough work to do. The body of the pressure kernel loop when using this method is shown in Listing 3. Rather than perform the computation on the entire domain before starting the exchange, the kernel is started with just the edges being computed. The first portion of the previously shown non-blocking MPI `EXCHANGE` operation is then started, which does device to host memory copy operations followed by non-blocking MPI communications. The computation on the middle portion of the domain can start as soon as the edge layers have finished transferring to the host, and operates in parallel with the MPI communication. The last part of the non-blocking MPI `EXCHANGE` operation is also identical and is run immediately after the middle computation is started. While this implementation results in significant overlap, it is possible to improve on it by overlapping the computation of the middle portion with the memory transfer of the edge layers as shown in the final implementation.

4.2.3. Overlapping Computation with MPI Communications and GPU Transfers

The final implementation is enabled by CUDA streams, and uses asynchronous methods to start the computation of the middle portion as soon as possible, thereby overlapping computation, memory operations, and MPI communication. A similar approach is described in Micikevicius [18]. This

```
// The GPU domain is decomposed into three sections:
//    (1) top edge, (2) bottom edge, and (3) middle
// Which of them the kernel should process is indicated
// by a flag given as an argument.

pressure <<<grid_edge,block>>> (edge_flags, div,p,pnew);

// The cudaMemcpy calls below will not start until
// the previous kernels have completed.
// This is identical to part 1 of the EXCHANGE operation.

// Communication to south
MPI_Irecv(new ghost layer from north)
cudaMemcpy(south edge layer from device to host)
MPI_Isend(south edge layer to south)
// Communication to north
MPI_Irecv(new ghost layer from south)
cudaMemcpy(north edge layer from device to host)
MPI_Isend(north edge layer to north);

pressure <<<grid_middle,block>>> (middle_flag, div,p,pnew);

// This is identical to part 2 of the EXCHANGE operation.
MPI_Wait(new ghost layer from north)
cudaMemcpy(new north ghost layer from host to device)
MPI_Wait(new ghost layer from south)
cudaMemcpy(new south ghost layer from host to device)
MPI_Waitall(south and north sends, allowing buffers to be reused)

pressure_bc <<<grid,block>>> (pnew);
ROTATE_POINTERS(p,pnew);
```

Listing 3. An example Jacobi pressure loop, showing how the CUDA kernel is split to overlap computation with MPI communication.

```
pressure <<<grid_edge,block, stream[0]>>> (edge_flags, div,p,pnew);
// Ensure the edges have finished before starting the copy
cudaThreadSynchronize();

cudaMemcpyAsync(south edge layer from device to host, stream[0])
cudaMemcpyAsync(north edge layer from device to host, stream[1])

pressure <<<grid_middle,block, stream[2]>>> (middle_flag, div,p,pnew);

MPI_Irecv(new ghost layer from north)
cudaStreamSynchronize(stream[0]);
MPI_Isend(south edge layer to south)

MPI_Irecv(new ghost layer from south)
cudaStreamSynchronize(stream[1]);
MPI_Isend(north edge layer to north);

MPI_Wait(south receive to complete)
cudaMemcpyAsync(new south ghost layer from host to device, stream[0])
MPI_Wait(north receive to complete)
cudaMemcpyAsync(new north ghost layer from host to device, stream[1])

// Ensure all streams are done, including copy operations and computation
cudaThreadSynchronize();
pressure_bc <<<grid,block>>> (pnew);
ROTATE_POINTERS(p,pnew);
```

Listing 4. CUDA streams are used to fully overlap computation, memory copy operations, and MPI communication in the pressure loop.

method has the highest amount over overlapping, and is expected to have the best performance at large scales. The body of the pressure kernel loop when using this method is shown in Listing 4.

It is important to note that the computations inside the CUDA kernels need minimal change, and the same kernel can be used for all three implementations. A flag is sent to each kernel to indicate which portions (top, bottom, middle) it is to compute, along with an adjustment of the CUDA grid size so the proper number of GPU threads are created. Since GPU kernels tend to be highly optimized, minimizing additional changes in kernel code is desirable.

4.3. Tri-Level MPI-OpenMP-CUDA Implementation

GPU cluster nodes are becoming denser with multiple GPUs per node [51]. Therefore we add a threading model to investigate whether additional efficiency can be gained from removing redundant message passing when processes are on the same host and communication and synchronization are handled by a hybrid MPI-OpenMP model. The effectiveness of this solution depends on a number of factors, with some barriers to effectiveness being:

- *Density of nodes.* With more GPUs per node, the potential effectiveness can be increased. Only clusters with two GPUs per node were available for the present study.

- *MPI implementation efficiency.* The OpenMPI 1.3.2 software on the NCSA Lincoln Tesla cluster seems reasonably well optimized. Goglin [52] discusses optimizations of MPI implementations to improve intra-node efficiency. A number of optimizations have been performed on MPI implementations since the early hybrid model papers were written, including a reduction in the number of copies involved. Since the application being studied only uses OpenMP and MPI for coarse-grain parallelism, any benefits in latency for small transactions will not have an impact.

- *A large number of nodes.* Many of the hybrid model papers note benefits occurring only as the number of nodes grows [26, 36, 38]. While the 64-node 128-GPU implementation used in this study is larger than many published cluster results, it may still be too small to see an appreciable benefit.

- *A good match between the hardware, the threading models, and the domain decomposition.* A number of hybrid model papers show application / hardware combinations that show reduced performance with the hybrid model [26, 28, 30, 35].

- Interactions between OpenMPI, OpenMP, and CUDA can exist. For instance, the default OpenMPI software on the NCSA Lincoln Tesla cluster is compiled without threading support.

There are two popular threading models in use today: POSIX Threads (Pthreads) and OpenMP. We consider OpenMP, because it has become the dominant method for shared memory parallelism in the HPC community. In our implementation the thread level parallelism is on a coarse grain level, since CUDA is handling the fine grain parallelism. We do not consider a more general approach where OpenMP can be used to perform some of the computations on multi-core CPUs in addition to computations on the GPU.

MPI defines four levels of thread safety: `SINGLE`, where only one thread is allowed. `FUNNELED` is the next level, where only a single master thread on each process may make MPI calls. The third level, `SERIALIZED`, allows any thread to make MPI calls, but only one at a time is using MPI. Finally, `MULTIPLE` allows complete multithreaded operation, where multiple threads can simultaneously call MPI functions.

With many clusters having pre-installed versions of MPI libraries, sometimes with custom network infrastructure, it is not always possible to have access to the highest (`MULTIPLE`) threading level. Additionally, this level of threading support typically comes with some performance loss, so lower levels are preferred if they do not otherwise hinder parallelism [53]. Three implementations were created, using the `SERIALIZED`, `FUNNELED`, and `SINGLE` levels. The first implementation used one thread per GPU, with each thread responsible for any possible MPI communications with neighboring nodes. The second used $N + 1$ threads for N GPUs, where a single thread per node handles all MPI communications and the other threads manage the GPU work. This can help alleviate resource contention between MPI and GPU copies, since each activity is on its own thread. Additionally this lets one use the `FUNNELED` level, which increases portability and possibly can increase performance. Lastly, the third version uses OpenMP directives to only perform MPI calls inside single-threaded sections.

Similar to the dual-level MPI-CUDA testing, simulation runs were performed on the NCSA Lincoln Tesla cluster for the tri-level parallel implemen-

```
// COMPUTE EDGES
if (threadid > 0)
  pressure <<<grid_edge,block>>> (edge_flags, div,p,pnew);

#pragma omp single
{
  MPI_Irecv(new ghost layer from north)
}
if (threadid > 0)
  cudaMemcpy(south edge layer from device to host)
// Ensure all threads have completed copies
#pragma omp barrier
#pragma omp single
{
  MPI_Isend(south edge layer to south)
  MPI_Irecv(new ghost layer from south)
}
if (threadid > 0)
  cudaMemcpy(north edge layer from device to host)
// Ensure all threads have completed copies
#pragma omp barrier
#pragma omp single
{
  MPI_Isend(north edge layer to north)
}

// COMPUTE MIDDLE
if (threadid > 0)
  pressure <<<grid_middle,block>>> (middle_flag, div,p,pnew);

#pragma omp single
{
  MPI_Wait(new ghost layer from north)
  MPI_Wait(new ghost layer from south)
}
// Ensure all threads wait for MPI communication
#pragma omp barrier
if (threadid > 0) {
  cudaMemcpy(new north ghost layer from host to device)
  cudaMemcpy(new south ghost layer from host to device)
}
// Ensure all threads have completed copies
#pragma omp barrier
#pragma omp single
{
  MPI_Waitall(south and north sends, allowing buffers to be reused)
}

if (threadid > 0)
  pressure_bc <<<grid,block>>> (pnew);

ROTATE_POINTERS(p,pnew);
```

Listing 5. An example Jacobi pressure loop using tri-level MPI-OpenMP-CUDA and simple computational overlapping. This uses the `SINGLE` threading level.

tation. At the time this study was performed, the MPICH2 implementation on NCSA Lincoln had interactions with the CUDA pinned memory support, making it very slow for the CUDA Streams overlapping cases. OpenMPI was used instead. But, unfortunately, the OpenMPI versions available on NCSA Lincoln do not support any threading level other than `SINGLE`, and optimal network performance was not obtainable with custom compiled versions by the first author. Hence only the last implementation was used. An example implementation is shown in Listing 5, where simple computational overlapping is performed. CUDA computations are performed on threads $1 - N$, while MPI calls are performed on the single thread 0. With a `FUNNELED` hybrid implementation, the `omp master` pragma would be used instead, with care taken since it has no implied barrier as `omp single` does.

4.4. Parallel Geometric Multigrid Method

Solution of complex incompressible flows benefits substantially from an advanced solver for the pressure Poisson equation, such as a multigrid (MG) method. The parallel geometric multigrid method that we implement in this study is built upon the strategies and lessons learned in previous sections. Based on the performance results obtained from parallel computations that adopt the Jacobi solver, we choose to follow the MPI-CUDA implementation described in section 4.2.3 in our MG method implementation. The 3D geometric MG method is composed of the *restriction, smoothing,* and *prolongation* steps. In the restriction step we use a 27-point full weighting scheme to restrict the residual solution from the fine grid to the next coarse grid level. The prolongation operator is the inverse operator of the restriction step. Therefore, we use a trilinear interpolation in the prolongation stage. In the smoothing stage, we use a weighted ($\omega = 0.86$) Jacobi solver with 3 to 4 iterations as the *smoother* for 3D computations.

Different schemes can be adopted to coarsen the grid in the MG method [56]. In our implementation, we use the V-cycle, which is adequate for the solution of pressure Poisson equation resulting from incompressible flow formulations. We develop an amalgamation strategy to overcome the data-starvation issue that arises in a multi-GPU implementation of the MG method. Basically, when the mesh at the finest level is divided and distributed over the GPUs, data-starvation per GPU is inevitable because of the inherent grid coarsening strategy in the MG method. When the coarsest grid per GPU is reached, the overall solution has not reached the deepest level in the V-cycle. We call the implementation that halts the grid coarsening process

when the coarsest mesh per GPU is reached as the *truncated* MG method. Depending on the size of the mesh and the number of GPUs deployed in the computations, truncating the MG cycle can substantially degrade the superior convergence rate of the MG method. To avoid this issue, we develop an amalgamation strategy to complete the V-cycle to its full-depth for the whole mesh. Our amalgamation strategy make use of the collective communication in the MPI library. Specifically, we use the `MPI Gather` function to reconstruct the mesh on a single GPU, and continue with the V-cycle down to its full-depth until the coarsest mesh for the overall domain is reached. Once the coarse grid solution is performed on a single GPU, we proceed with the V-cycle on a single GPU and scatter the information to all GPUs with an `MPI_Scatter` function at the same MG level where the amalgamation to a single GPU took place. The amalgamation strategy enables us to achieve the superior efficiency of the MG method in a parallel multi-GPU implementation.

5. Performance Results from NCSA Lincoln and TACC Longhorn Clusters

The NCSA Lincoln cluster consists of 192 Dell PowerEdge 1950 III servers connected via InfiniBand SDR (single data rate) [54]. Each compute node has two quad-core 2.33 GHz Intel E5410 processors and 16GB of host memory. The cluster has 96 NVIDIA Tesla S1070 accelerator units each housing four C1060-equivalent Tesla GPUs. An accelerator unit is shared by two servers via PCI-Express $\times 8$ connections. Hence, a compute-node has access to two GPUs. For the present study, performance measurements for 64 of the 192 available compute-nodes in the NCSA Lincoln Tesla cluster are shown, with up to 128 GPUs being utilized. The CUDA 3.0 Toolkit was used for compilation and runtime, gcc 4.2.4 was the compiler used, and OpenMPI 1.3.2 was used for the MPI library.

The TACC Longhorn cluster consists of 240 Dell R610 compute nodes connected via InfiniBand QDR (quad data rate). Each compute node has two quad-core 2.53 GHz Intel E5540 Nehalem processors and 48GB of host memory. The cluster has 128 NVIDIA QuadroPlex S4 accelerator units each housing four FX5800 GPUs. An accelerator unit is shared by two servers via PCI-Express 2.0 $\times 16$ connections. Performance of the GPU units is similar to the Lincoln cluster, however the device/host memory bandwidth is more than $2\times$ higher and the cluster interconnect is $4\times$ faster. For the present study,

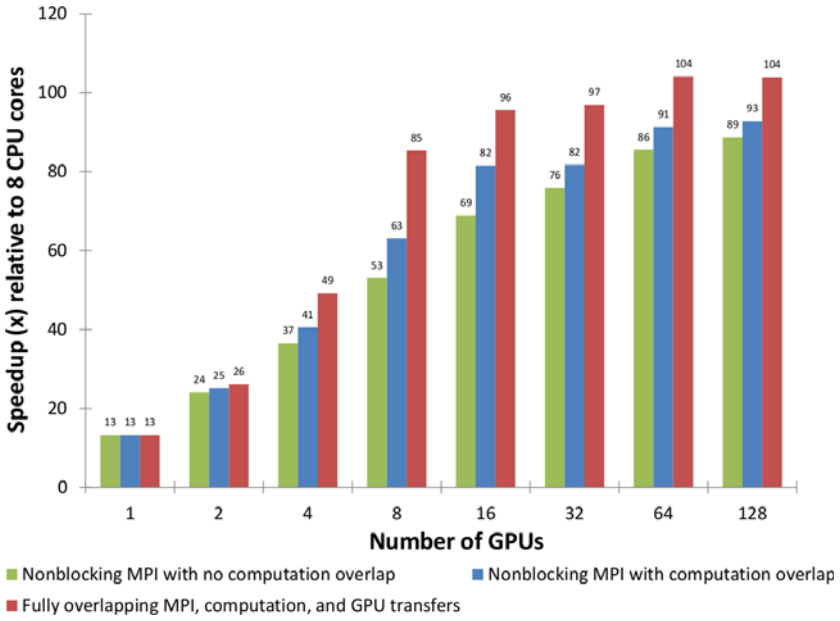


Figure 5. Speedup on the NCSA Lincoln Tesla cluster from the three MPI-CUDA implementations relative to the Pthreads parallel CPU code using all 8 cores on a compute-node. The lid-driven cavity problem is solved on a $1024 \times 64 \times 1024$ grid with fixed number of iterations and time steps.

performance measurements for 128 of the 240 available compute-nodes in the TACC Longhorn cluster are shown, with up to 256 GPUs being utilized. The CUDA 3.0 Toolkit was used for compilation and runtime, gcc 4.1.2 was the compiler used, and OpenMPI 1.3.3 was used for the MPI library.

Single GPU performance has been studied relative to a single CPU processor in many studies. Such performance comparisons are adequate for desktop GPU platforms. On a multi-GPU cluster, a fair comparison should be based on all the available CPU resources in the cluster. To partially address this issue, the CPU version of the CFD code is parallelized with Pthreads to use the eight CPU cores available on a single compute-node of the NCSA Lincoln cluster [15, 16]. Identical numerical methods are used in the CPU and GPU code for the tests performed. In Thibault and Senocak [16], the performance of the CPU version of the code was investigated and the GFLOPS performance was found to be comparable to the NPB benchmark codes.

A lid-driven cavity problem at a Reynolds number of 1000 was chosen for performance measurements. Measurements were performed for both *strong scaling* where the problem size remains fixed as the number of processing elements increases, and *weak scaling* where the problem size grows in direct proportion to the number of processing elements. Measurements for the CPU application were done using the Pthreads shared-memory parallel implementation using all eight CPU cores on a single compute-node of the NCSA Lincoln cluster. All measurements include the complete time to run the application including setup and initialization, but do not include I/O time for writing out the results. Single precision was used in all computations.

Strong Scaling Analysis

Figure 5 shows the speedup of the MPI-CUDA implementation of our flow solver relative to the performance of the CPU version of our solver using Pthreads. The computational performance on a single compute-node with 2 GPUs was 26× faster than 8 Intel Xeon cores, and 64 compute-nodes with 128 GPUs performed up to 104× faster than 8 Intel Xeon cores. In all configurations the fully overlapped implementation performed faster than the first implementation that did not perform overlapping. Additionally, the final fully overlapping implementation performs fastest in all configurations with more than one GPU, and shows a significant benefit with more than four GPUs. With the fixed problem size, the amount of work to do on each node quickly drops — on a single GPU a single pressure iteration takes under 10ms of compute time. Little gain is seen beyond 16 GPUs on this fixed

size problem, which highlights the fact that GPU clusters problems with large data sets.

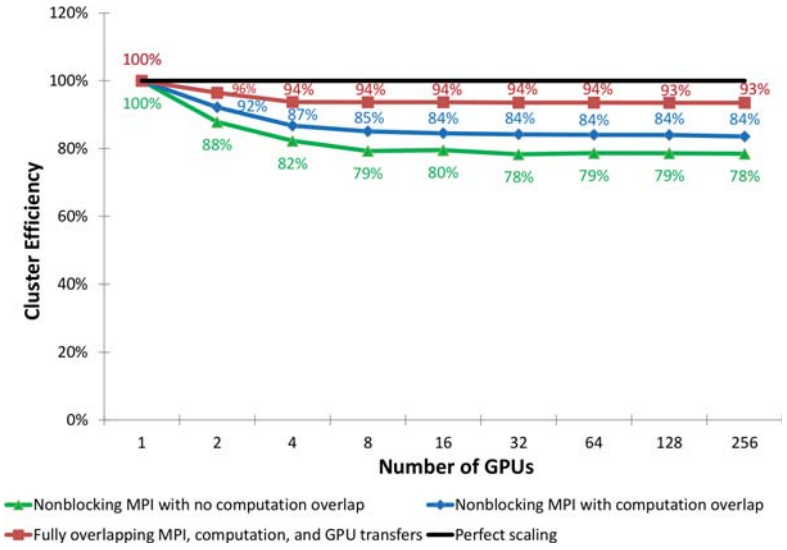
Weak Scaling Analysis

All three MPI-CUDA implementations presented in section 4.2 were also run with increasing problem sizes such that the memory used per GPU was approximately equal. The analysis is commonly referred to as *weak scalability*. Simulations such as channel or duct flows can lead to extension of the whole domain in one of the three dimensions as the problem size increases. In this case the height and depth of a channel is fixed, while the width increases relative to the number of GPUs. For the 1D network decomposition performed in our flow solver, the amount of data transferred between each GPU will be constant, as will the domain dimensions on each GPU. Therefore we expect the scalability to be excellent.

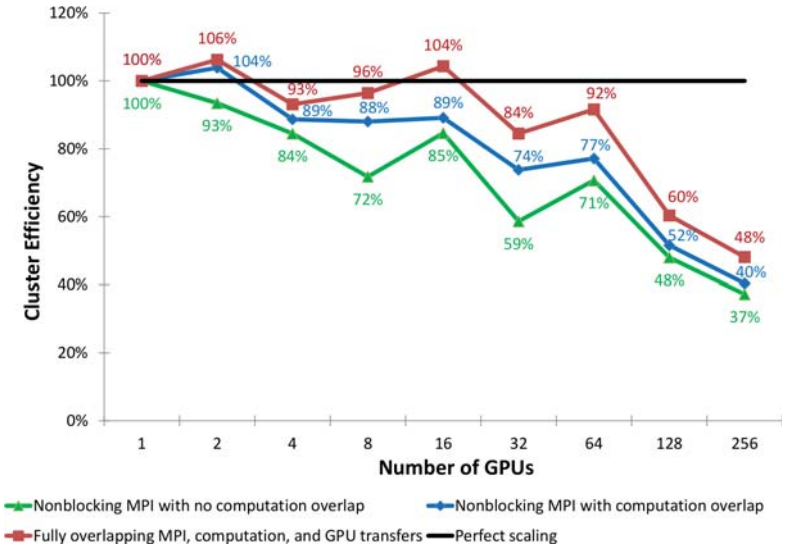
Figure 6a indicates how scalability with the fully overlapped implementation performs so well in this one dimensional scaling case, dropping from 94% with 4 GPUs to only 93% with 128 GPUs. Note that four GPUs is the first case where the network is utilized. The results from the TACC Longhorn cluster shows a consistent behavior, with only a 1% drop in efficiency from 4 GPUs to 256 GPUs. The fully overlapped MPI-CUDA implementation shows a definite advantage over the other two MPI-CUDA implementations.

Figure 6b shows the parallel efficiency when the computational domain grows in two dimensions during a weak scaling analysis. This is a very common scenario seen in such examples as many lid-driven cavity and buoyancy-driven cases, as well as flow in complex terrain, where covering a larger physical area (e.g. more square blocks in an urban simulation) involves growth in the horizontal dimensions, while the number of cells used for height remains constant. On the TACC Longhorn cluster, 256 GPUs were utilized on 128 compute-nodes to sustain an 4.9 TeraFLOPS performance. With approximately 400GB of memory used during the computation on 128 GPUs, it is not possible to directly compare this to a single node CPU implementation on traditional machines. Figure 6b also shows the clear advantage of overlapping computation and communication. Parallel efficiency in the two-dimensional growth problem with full overlapping is excellent through 64 GPUs, and parallel efficiency drops to 60% beyond 64 GPUs.

One obvious feature of Figure 6(b) is that efficiency does not fall in a smooth fashion with increasing GPUs, but steps up and down with an



a) 1D Growth



b) 2D Growth

Figure 6. Efficiency of the three MPI-CUDA implementations with increasing number of GPUs on the TACC Longhorn cluster (weak scalability presentation). a) Growth is in one dimension. The size of the computational grid is varied from $512 \times 512 \times 256$ to $512 \times 512 \times 65536$ with increasing number of GPUs. b) Growth is in two dimensions, with the Y dimension fixed. The size of the computational grid is varied from $1024 \times 64 \times 1024$ to $16384 \times 64 \times 16384$ with increasing number of GPUs. Using 256 GPUs,computations sustained 8.5 TeraFLOPS in 1D the growth case and 4.9 TeraFLOPS in the 2D growth case.

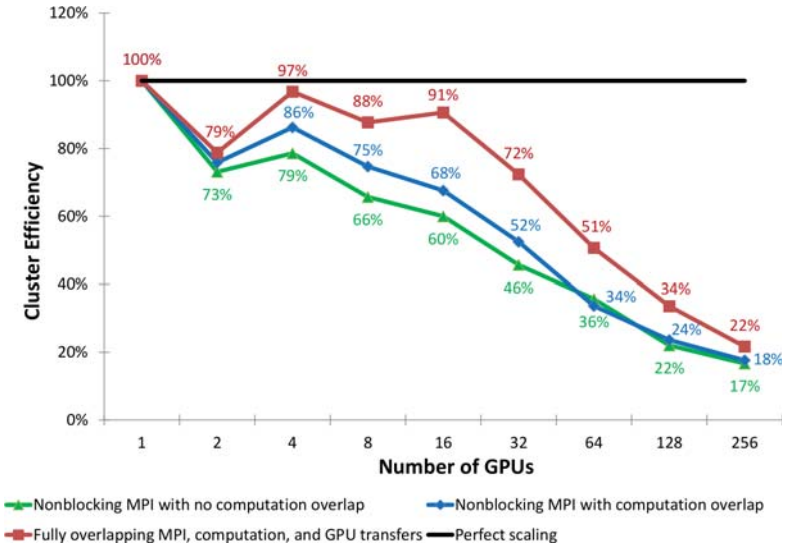
overall decreasing trend. This is related to an interaction between the two-dimensional problem size growth and the structure of the CUDA kernels. The mechanism of having each thread loop over all the Z planes is very efficient, however the CUDA kernel throughput strongly changes as the numbers of threads (the X and Y dimensions) relative to the number of Z planes per GPU is varied. Earlier implementations of the kernels, as seen in Jacobsen et al. [55], show much less variability, but overall performance is lower — for a similar problem the single GPU performance is 33 GFLOPS vs. 41 for the current code, and 2.4 TFLOPS vs. 2.9 TFLOPS with 128 GPUs.

Figure 7 presents the weak scaling analysis for a growth in three dimensions of the computational domain on the Longhorn cluster. Figure 7(a) indicates how scalability with the fully overlapped implementation trails off sharply at 16 GPUs, and the gap between the overlapping implementations and non-overlapping narrows. The reasons for this behavior are examined in the next section. Figure 7(b) shows the sustained GFLOPS performance on a logarithmic scale. With 256 GPUs, 2.4 TeraFLOPS was sustained with the fully overlapped implementation. Note that for the 1D growth case, 9.5 TeraFLOPS was sustained using the same number of GPUs.

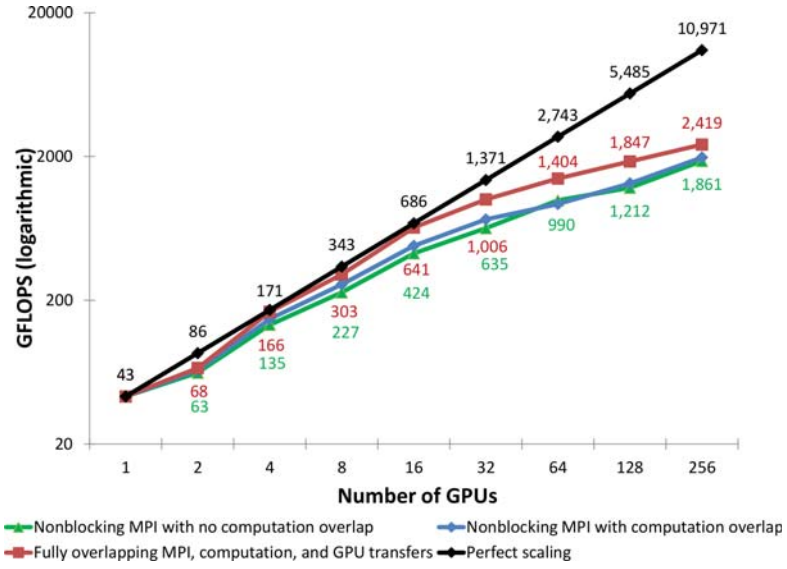
Further Remarks on Scalability

NCSA Lincoln cluster was transformed into a GPU cluster from an existing CPU cluster. The connection between the compute-nodes and the Tesla GPUs are through PCI-Express Gen 2 $\times 8$ connections rather than $\times 16$. Measured bandwidth for pinned memory is approximately 1.6 GB/s, which is significantly slower than the 5.6 GB/s measured on a local workstation with PCIe Gen 2 $\times 16$ connections to Tesla C1060s. Kindratenko et al. [54] observed a low host-device bandwidth on Lincoln cluster, and suggested further investigations. This observed low-bandwidth issue with the Lincoln cluster has an impact on our results.

We performed bandwidth measurements on the TACC Longhorn cluster which uses GPUs with similar performance (Quadroplex 2200 S4 on Longhorn, Tesla S1070 on Lincoln). However, measured device/host memory transfers are over $2\times$ faster on Longhorn, and its Infiniband QDR shows a $4\times$ increase in interconnect bandwidth with simple benchmarks. It should also be pointed out that as the CUDA kernels are optimized and run faster, less time becomes available for overlapping communications, leading to a loss in parallel efficiency.

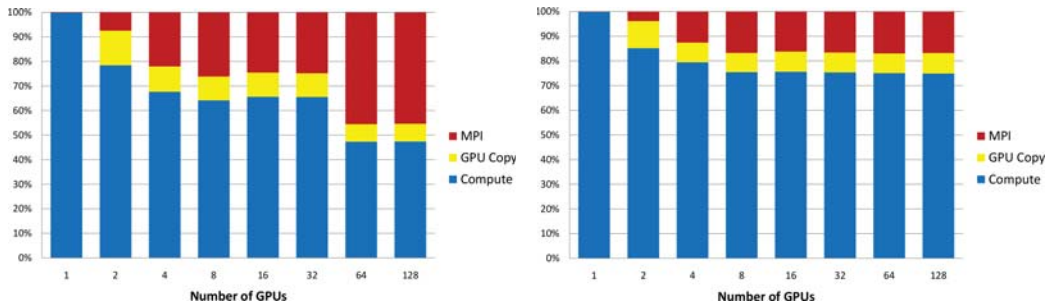


a) Cluster efficiency

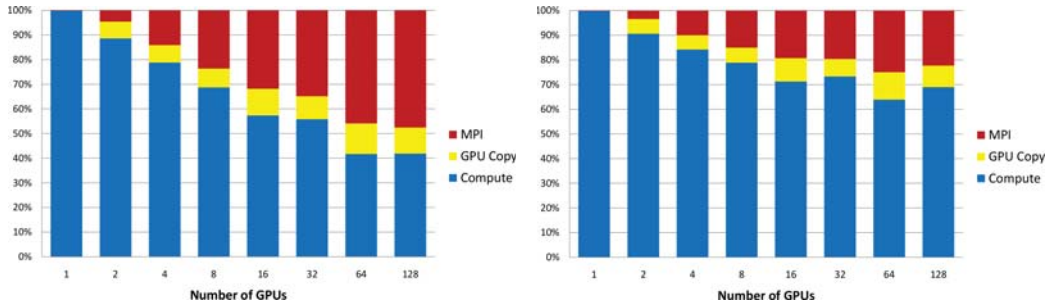


b) Performance in GFLOPS

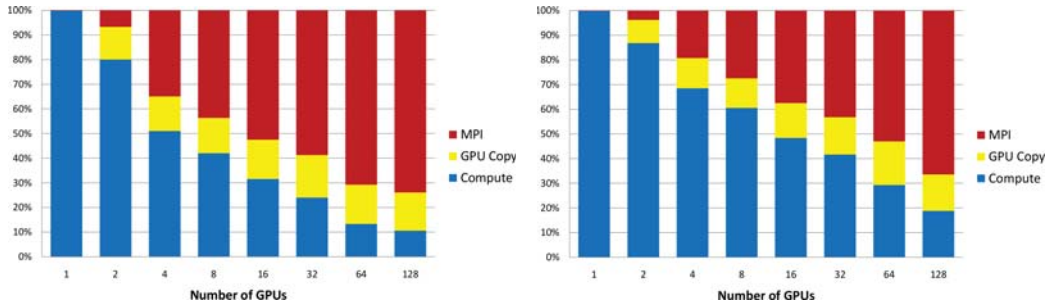
Figure 7. Efficiency of the three MPI-CUDA implementations with increasing number of GPUs on the TACC Longhorn cluster (weak scalability presentation). Growth is in three dimensions. The size of the computational grid is varied from $416 \times 416 \times 416$ to $2688 \times 2688 \times 2560$ with increasing number of GPUs. a) Parallel cluster efficiency, b) Perfomance in GFLOPS.



a) 1D Growth (Lincoln) b) 1D Growth (Longhorn)



c) 2D Growth (Lincoln) 2D Growth (Longhorn)



e) 3D Growth (Lincoln) f) 3D Growth (Longhorn)

Figure 8. Percent of pressure Poisson solver (30 Jacobi iterations) time spent in computation, host/GPU memory transfer, and MPI calls. No overlapping is used. The problem size grows such that the number of cells per GPU is approximately constant. a–b) 1D growth, c–d) 2D growth, e–f) 3D growth.

To further examine the reasons behind the scalability results seen, CUDA event timers were used to get high resolution profiles of time spent in the iterative pressure solver. The timers calculated the time spent doing computation in the pressure and boundary condition kernels, the amount of time spent copying data between the GPU and the host, and the time spent in network communications. This data was collected using the implementation described in section 4.2.1, to show the need for overlapping as well as shed light into the earlier scalability graphs.

For the 1D growth case shown in Figure 8a–b, measured compute and GPU copy time was essentially constant for all runs. This is expected, as the per-GPU dimensions of the pressure domain are identical at each size, and the amount of data to be transferred is constant. The amount of data exchanged by each host also remains constant as the number of GPUs increases, yet the time spent in MPI calls on the Lincoln cluster increases with more GPUs. While performing the solver iterations, each process only synchronizes with its immediate neighbors – no global operations are used. We attribute the observed behavior as a network topology issue with the Lincoln cluster and not with our implementation because it is absent in the results using the Longhorn cluster. On the Longhorn cluster, as shown in Figure 8b, the percent of time spent in copy and MPI is essentially constant once the network is utilized at 4 GPUs, which is what is expected.

With the 2D growth case shown in Figure 8c–d, the amount of data to be transferred grows by a factor of $\sqrt{N}$ as the number of GPUs (N) increases. In the 4 GPU case each transferred layer consists of 2048×64 cells, while with 16 GPUs (a $4\times$ increase) each layer has 4096×64 cells — a $2\times$ increase. With 32 or fewer GPUs, it is possible to completely overlap network traffic and GPU copies with computation. However, the particular size used in this simulation for 32 and 128 GPUs leads to slower computation than other cases, as remarked upon earlier to explain the wiggly trend in parallel efficiency in Figure 6b. With 64 and 128 GPUs, complete overlapping of copy, MPI, and computation needs to be done to keep scalability. The data on Longhorn shows a similar pattern, yet scales better as the communication paths are faster.

The 3D growth case is shown in Figure 8e–f. The amount of data to be transferred grows by a factor of $N^{2/3}$ with the number of GPUs. In the single GPU case each transferred layer consists of 416×416 cells, while with 64 GPUs each layer has 1664×1664 cells — a $16\times$ increase. Both the GPU copy and MPI communication time increase rapidly, with the GPU copy

alone taking more time on 64 GPUs than the entire computation time. The picture on the Longhorn cluster is similar, with the faster data copies just moving the saturation point to more GPUs. While large linear transfers are done to achieve maximum copy efficiency, the amount of data is too large in these cases. Calculations are shown below for the 64 GPU case on the Lincoln cluster:

$$\begin{aligned} Copy\ Bandwidth &= (layer\ size \cdot 4 \cdot iterations \cdot timesteps)\,/time \qquad (5)\\ &= ((1664 \times 1664 \times 4 \text{ bytes}) \cdot 4 \cdot 30 \cdot 200)\,/139.62 \text{ seconds}\\ &= 1816 \text{ MB/s} \end{aligned}$$

$$\begin{aligned} MPI\ Bandwidth &= (layer\ size \cdot 4 \cdot iterations \cdot timesteps)\,/time \qquad (6)\\ &= ((1664 \times 1664 \times 4 \text{ bytes}) \cdot 4 \cdot 30 \cdot 200)\ /\ 624.5 \text{ seconds}\\ &= 405.9 \text{ MB/s} \end{aligned}$$

For each GPU, the two edge layers must be copied from the GPU and then again to the GPU, hence the factor of 4. This simple calculation ignores the effect of the edge nodes. The effective GPU copy bandwidth is similar to that reported with memory benchmarks on this platform, which is 2 to 3 times less than newer hardware. The effective MPI bandwidth is lower than the bidirectional bandwidth measured with MPI benchmarks, suggesting this as a possible point to investigate.

A 2D decomposition would greatly reduce the amount of data transferred with these large 2D and 3D simulations. Assuming a domain partition in the growth dimensions, the 2D and 3D simulations would see a 4× reduction in the number of bytes transferred. The ramifications to CUDA are discussed in section 4.1. It is likely that for 3D problems on many GPUs, the extra CUDA work may be worth the per-GPU cost.

Figure 9 directly compares the weak scaling efficiency with growth in three dimensions using a fully overlapped version of our flow solver on NCSA Lincoln and TACC Longhorn clusters. While the improved communication bandwidth on the TACC Longhorn cluster greatly helps scalability (at 128 GPUs, Lincoln is at 13% while Longhorn achieves 34%), the overall trend in weak scaling is similar. On the NCSA Lincoln Tesla cluster, only 768 GFLOPS was sustained with the fully overlapped implementation using 128

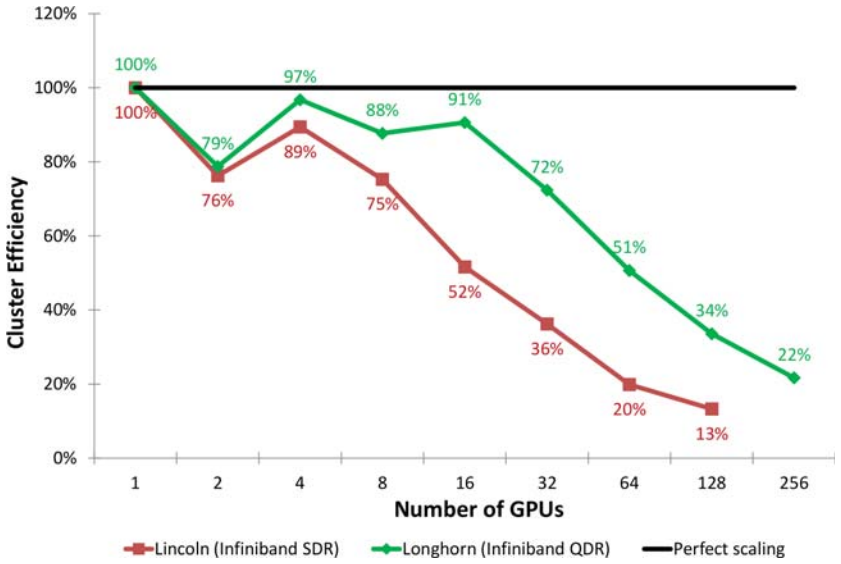


Figure 9. A comparison of weak scaling with the fully overlapped MPI-CUDA implementation on two platforms, with growth in three dimensions. Longhorn has higher bandwidth for both GPU/host and network data transfer than Lincoln.

GPUs. On the TACC Longhorn cluster, 2.4 TeraFLOPS was sustained using 256 GPUs.

Performance Analysis of Tri-level MPI-OpenMP-CUDA Implementation

Similar to the dual-level performance results, a lid-driven cavity problem at a Reynolds number of 1000 was chosen for performance measurements on the NCSA Lincoln Tesla cluster. As mentioned in section 4.3 earlier, software issues on the NCSA Lincoln cluster precluded effective testing of anything but the tri-level implementation to use single threading. The weak scaling analysis with growth in three dimensions is the most taxing case on cluster efficieny as compared to growth in one and two dimensions, and shows the most difference between the parallel methods considered. Therefore we evaluate the tri-level parallel implementation using weak scaling analysis with growth in three dimensions, and compare it against the best performing dual-level parallel implementation.

Figure 10 compares the the scaling efficiency of the fully overlapped dual-level MPI-CUDA and the tri-level MPI-OpenMP-CUDA implementations in

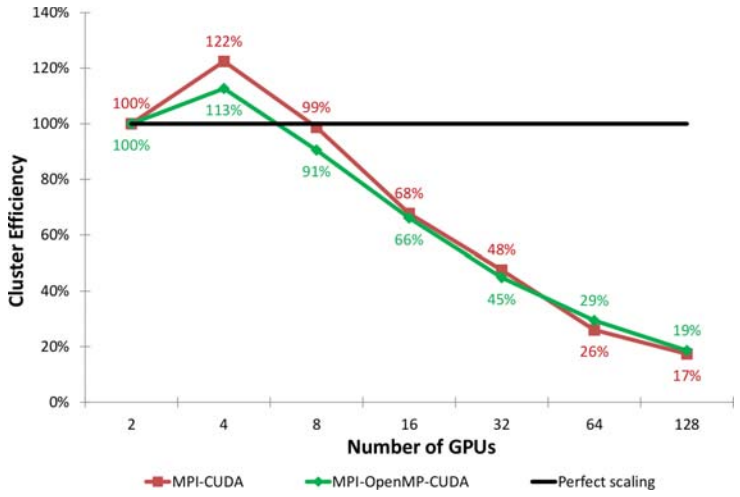


Figure 10. A comparison of weak scaling with the fully overlapped MPI-CUDA and single threaded MPI-OpenMP-CUDA implementations, with growth in three dimensions. Since the tri-level implementation uses all the GPUs of a single node, the base value for parallel scaling is set to a single node of the NCSA Lincoln Tesla cluster containing two GPUs.

the 3D growth weak scaling scenario. The MPI-CUDA data matches the fully overlapped data from Figure 7, though 100% is set with two GPUs (a single node) rather than one. We decided to calculate the cluster efficiency relative to the performance of two GPUs in this particular case, because tri-level implementation uses all the GPUs of single node with OpenMP addressing the intra-node parallelism and MPI handling the inter-node parallelism. Hence, the super-efficiency observed at 4 GPUs is direct outcome of how we calculate the parallel efficiency in this particular case.

With fewer than 4 nodes (8 GPUs), the dual-level MPI-CUDA implementation performs better. With 32 and 64 nodes (64 and 128 GPUs), there is a small benefit with the present MPI-OpenMP-CUDA implementation. At this point the amount of data being transferred may bring any efficiencies of the shared memory model to the forefront, outweighing single-node synchronization. Our results are consistent with the hybrid performance results shown by Nakajima [38], where MPI-vector implementation outperformed the hybrid MPI-OpenMP-vector implementation at 64 and fewer nodes, and started showing an increasing benefit at 96 nodes and and beyond. We were not able to measure the results beyond 64 nodes (128 GPUs), but we believe

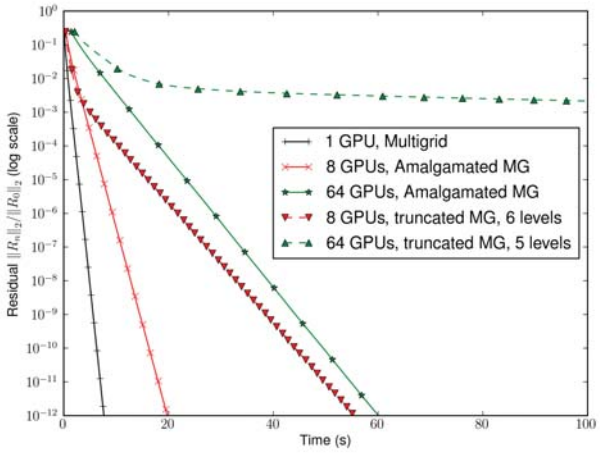


Figure 11. Performance and parallel efficiency of the V-cycle truncated and amalgamated multigrid on 1, 8, and 64 GPUs where the problem size scales with the number of GPUs. Time is plotted against the residual level for a double precision problem using 257^3 on 1 GPU, 513^3 using 8 GPUs, and 1025^3 using 64 GPUs on the NCSA Lincoln Tesla cluster. A marker is shown for each 4 loops of the multigrid cycle.

the performance of the tri-level implementation should be further investigated on larger clusters with more than two GPUs per node and also with different domain decomposition strategies.Unfortunately, such large clusters with dense GPU nodes were not available or accessible during our study.

Performance of the Parallel Geometric Multigrid Method

A 3D lid-driven cavity problem was started at grid sizes of 257^3, 513^3 and 1025^3 using 1, 8 and 64 GPUs on the NCSA Lincoln Tesla cluster. We used double precision in all computations. The actual wall-time taken by the pressure solver is plotted against the residual level for the initial time step.

Figure 11 shows the performance of the multigrid algorithm on the NCSA Lincoln Tesla cluster for relatively large problems (16M, 128M, and 1024M cells). In particular the results of the amalgamated full-depth multigrid are compared to the truncated multigrid, and the single-GPU multigrid implementation. We note that on a single GPU, issues of amalgamation and incomplete V-cycles are absent. With 8 GPUs, the coarsest grid is 17^3, while

with 64 GPUs with coarsest grid is 65^3. On a single GPU a full-depth V-cycle was performed, hence there is no truncation of the V-cycle. Our results show the clear benefit of amalgamation on the convergence rate of a multi-GPU implementation of the MG method. At the larger problem sizes, the convergence rate of the truncated multigrid is unacceptable and the need for amalgamation in the parallel multigrid method becomes obvious.

The multigrid level at which amalgamation to a single GPU takes place has an effect on the performance. The current implementation can amalgamate to a single GPU at the third level in the V-cycle for most problems considered in this study. However, for a grid size of 1025^3 we found that amalgamating at the fourth level or deeper levels produces same performance results, and they are better than performance results obtained when amalgamating at the third level. We note that the level at which to amalgamate depends on computational problem and device memory sizes.

6. Conclusions

We have presented both dual-level (MPI-CUDA) and a tri-level (MPI-OpenMP-CUDA) parallel implementations of a Navier-Stokes equations solver to simulate buoyancy-driven incompressible fluid flows on GPU clusters. We adopt NVIDIA's CUDA programming model for fine-grain data-parallel operations within each GPU. In the tri-level implementation we use OpenMP for intra-node communications within a compute-node, and MPI for communications across the cluster. In the dual-level implementation, MPI handles all intra- and inter-node communications.

We adopted a simple point iterative scheme to solve the pressure Poisson equation to investigate the interplay of computation, communications, and synchronizations in multi-level parallel implementations on a GPU cluster with different strategies to overlap computation with communications. However, many applications, including the present one, require advanced numerical methods and fast solvers such as the multigrid method. Therefore, we extended the best performing multi-level parallel implementation described in this study to a geometric multigrid method, in which we introduced an amalgamation strategy to recover the superior convergence rate of the multigrid method on GPU clusters.

In all the multi-level implementations we adopted a 1D domain decomposition strategy as the overhead for gathering and scattering the data into linear transfer buffers can exceed the advantages of the smaller transfer sizes

that one could get from 2D or 3D domain decompositions. An additional level of 1D domain decomposition is also introduced within the compute-space of each GPU to overlap intra- and inter-node data exchanges with advanced features of MPI and CUDA. We implemented three strategies to overlap computation with communications. With measurements from two different GPU clusters, we showed that performance and efficiency critically depends on the bandwidth of the network, and the strategy that introduces maximum overlapping of computation with communication improves the parallel performance markedly. Although we have used as many as 256 GPUs on 128 nodes of the Longhorn cluster with Infiniband QDR network, the parallel efficiency dropped below 50% beyond 64 GPUs on 32 nodes during weak scaling analysis with 3D growth in computational domain sizes, suggesting that multi-GPU computing can benefit substantially from advances in fast networking hardware.

Our performance measurements indicate that the dual-level (MPI-CUDA) parallel model with maximum overlapping produces the best performance. We believe the gain from the tri-level MPI-OpenMP-CUDA parallel method is unlikely to offset the additional software complexity that is introduced into the flow solver. Models that share fine-grain parallelism on multi-core CPUs with GPUs, a different domain decomposition strategy than is presented here or have high GPU density per node may see better results and need to be investigated further.

A number of issues with obtaining the most benefit from tri-level MPI-OpenMP-CUDA parallel methods have been identified. Compared to early published results, current MPI libraries have much better optimization for multiple processes per node. A number of the benefits ascribed to the hybrid MPI-OpenMP programming model are typically obtained via OpenMP's fine-grain parallelism support, which is not used at all in this study, because all fine-grain parallelism is supplied by CUDA. Other simulation software that can use both CPU and GPU resources for computation may show more advantage from tri-level parallelism. It is also an open question whether a much denser per-node GPU density may be able to take better advantage of the tri-level parallelism. We think having only two GPUs per node on current and planned GPU cluster designs puts a limit on the possible benefit from the mixed API model. At the time of the present study, GPU clusters with denser nodes were not available.

Finally, with our best performing implementation using 256 GPUs on the TACC Longhorn cluster, we were able to process 17 billion elements with

8.5, 4.9, 2.4 TeraFLOPS of single precision sustained performance in 1D, 2D and 3D growth during weak scaling analysis, respectively. On the NCSA Lincoln cluster, we have shown that 2-GPU performance of our solver is 26× faster than the 8-core CPU performance. Our results demonstrate that GPU clusters are powerful computing platforms to solve computationally large problems. With their heterogeneous architectures that can support both CPU and GPU based applications and graphics rendering, we expect a wide adoption of GPU clusters in the industry and academia.

Acknowledgments

This work is partially funded by grants from NASA Idaho Space Grant Consortium and National Science Foundation (NSF) (Award #1043107 and #1056110). We utilized the Lincoln Tesla Cluster at the National Center for Supercomputing Applications and the Longhorn visualization cluster at the Texas Advanced Computing under grant number ASC090054 and ATM100032 from NSF TeraGrid. We thank NVIDIA Corporation for hardware donations and extend our thanks to Marty Lukes of Boise State University for his continuous help with building and maintaining our GPU computing infrastructure.

References

[1] NVIDIA, NVIDIA CUDA Programming Guide 3.1.1, 2010.

[2] Khronos OpenCL Working Group, The OpenCL Specification: Version 1.1, 2010.

[3] M. Showerman, J. Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, W.-M. Hwu, QP: A heterogeneous multi-accelerator cluster, in: Proceedings of the 10th LCD International Conference on High-Performance Clustered Computing, Boulder, Colorado, 2009.

[4] NCSA, Intel 64 Tesla Linux cluster Lincoln webpage, `http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/` (2008).

[5] HPCwire, New Chinese GPGPU super outruns Jaguar, `http://www.hpcwire.com/blogs/New-China-GPGPU-Super-Outruns-Jaguar-105987389.html` (Oct. 2010).

[6] HPCwire, NVIDIA Tesla GPUs power world's fastest supercomputer, http://www.hpcwire.com/offthewire/NVIDIA-Tesla-GPUs-Power-Worlds-Fastest-Supercomputer-105983244.html (Oct. 2010).

[7] HPCwire, China's new Nebulae supercomputer is no. 2, right on the tail of ORNL's Jaguar http://www.hpcwire.com/offthewire/Chinas-New-Nebulae-Supercomputer-Is-No-2-Right-on-the-Tail-of-ORNLs-Jaguar-95258669.html (May 2010).

[8] H. Simon, T. Zacharia, R. Stevens, Modeling and simulation at the exascale for energy and the environment, Tech. rep., DOE ASCR Program (2008).

[9] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, T. J. Purcell, A survey of general-purpose computation on graphics hardware, Computer Graphics Forum 26 (1) (2007) 80–113.

[10] T. Brandvik, G. Pullan, Acceleration of a 3D Euler solver using commodity graphics hardware, in: 46th AIAA Aerospace Sciences Meeting and Exhibit, 2008.

[11] E. Elsen, P. LeGresley, E. Darve, Large calculation of the flow over a hypersonic vehicle using a GPU, Journal of Computational Physics 227 (24) (2008) 10148–10161.

[12] D. Göddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, H. Wobker, C. Becker, S. Turek, Using GPUs to improve multigrid solver performance on a cluster, International Journal of Computational Science and Engineering 4 (1) (2008) 36–55.

[13] J. M. Cohen, M. J. Molemaker, A fast double precision CFD code using CUDA, in: Proceedings of Parallel CFD, 2009.

[14] H. Schive, Y. Tsai, T. Chiueh, GAMER: a GPU-accelerated adaptive mesh refinement code for astrophysics, Astrophysical Journal Supplement Series 186 (2010) 457–484.

[15] J. C. Thibault, I. Senocak, CUDA implementation of a Navier–Stokes solver on multi-GPU platforms for incompressible flows, in: 47th AIAA Aerospace Science Meeting, 2009.

[16] J. C. Thibault, I. Senocak, Accelerating incompressible flow computations with a Pthreads-CUDA implementation on small-footprint multi-GPU platforms, The Journal of Supercomputing 59 (2) (2012) 693–719.

[17] D. A. Jacobsen, Methods for multilevel parallelism on GPU clusters: Application to a multigrid accelerated Navier-Stokes Solver, Master's thesis, Boise State University, Boise, Idaho (May 2011).

[18] P. Micikevicius, 3D finite difference computation on GPUs using CUDA, in: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009, pp. 79–84.

[19] D. Göddeke, S. H. Buijssen, H. Wobker, S. Turek, GPU acceleration of an unmodified parallel finite element Navier–Stokes solver, in: International Conference on High Performance Computing & Simulation, 2009, pp. 12–21.

[20] J. C. Phillips, J. E. Stone, K. Schulten, Adapting a message-driven parallel application to GPU-accelerated clusters, in: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 2008.

[21] E. H. Phillips, Y. Zhang, R. L. Davis, J. D. Owens, Rapid aerodynamic performance prediction on a cluster of graphics processing units, in: Proceedings of the 47th AIAA Aerospace Sciences Meeting, 2009.

[22] G. Krawezik, Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors, in: Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, San Diego, California, USA, 2003, pp. 118–127.

[23] M. J. Berger, M. J. Aftosmis, D. D. Marshall, S. M. Murman, Performance of a new CFD flow solver using a hybrid programming paradigm, J. Parallel Distrib. Comput. 65 (4) (2005) 414–423.

[24] F. Cappello, O. Richard, D. Etiemble, Performance of the NAS benchmarks on a cluster of SMP PCs using a parallelization of the MPI programs with OpenMP, in: PaCT, 1999, pp. 339–350.

[25] F. Cappello, O. Richard, D. Etiemble, Investigating the performance of two programming models for clusters of SMP PCs, in: HPCA, 2000, pp. 349–359.

[26] F. Cappello, D. Etiemble, MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks, in: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), IEEE Computer Society, Dallas, Texas, United States, 2000, p. 12.

[27] S. W. Bova, C. P. Breshears, C. E. Cuicchi, Z. Demirbilek, H. A. Gabb, Dual-level parallel analysis of harbor wave response using MPI and OpenMP, Int. J. High Perform. Comput. Appl. 14 (1) (2000) 49–64.

[28] D. S. Henty, Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling, in: Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing, IEEE Computer Society, Dallas, Texas, United States, 2000, p. 10.

[29] P. Luong, C. P. Breshears, L. N. Ly, Coastal ocean modeling of the U.S. west coast with multiblock grid and dual-level parallelism, in: Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing, ACM, New York, NY, USA, 2001, pp. 9–9. doi:http://doi.acm.org/10.1145/582034.582043.

[30] S. Dong, G. E. Karniadakis, Dual-level parallelism for deterministic and stochastic CFD problems, in: Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing, IEEE Computer Society Press, Los Alamitos, CA, USA, 2002, pp. 1–17.

[31] K. Nakajima, H. Okuda, Parallel iterative solvers for unstructured grids using an OpenMP/MPI hybrid programming model for the GeoFEM platform on SMP cluster architectures, in: Proceedings of the 4th International Symposium on High Performance Computing, Springer-Verlag, Kansai Science City, Japan, 2002, pp. 437–448.

[32] R. Rabenseifner, Communication bandwidth of parallel programming models on hybrid architectures, in: Proceedings of the 4th International Symposium on High Performance Computing, Springer-Verlag, Kansai Science City, Japan, 2002, pp. 401–412.

[33] R. Rabenseifner, Hybrid parallel programming on HPC platforms, in: EWOMP '03: Proceedings of the Fifth European Workshop on OpenMP, Aachen, Germany, 2003, pp. 185–194.

[34] A. Prabhakar, V. Getov, Performance evaluation of hybrid parallel programming paradigms, in: Performance analysis and grid computing, Kluwer Academic Publishers, 2004, pp. 57–76.

[35] E. Lusk, A. Chan, Early experiments with the OpenMP/MPI hybrid programming model, Lecture Notes in Computer Science 5004 (2008) 36.

[36] G. Hager, G. Jost, R. Rabenseifner, Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes, in: Proceedings of the Cray Users Group Conference, 2009.

[37] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. L. Lusk, R. Thakur, J. L. Träff, MPI on a million processors, in: Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2009, pp. 20–30.

[38] K. Nakajima, Three-level hybrid vs. flat MPI on the Earth Simulator: Parallel iterative solvers for finite-element method, Applied Numerical Mathematics 54 (2) (2005) 237–255.

[39] P. K. Kundu, I. M. Cohen, Fluid Mechanics, 4th Edition, Academic Press, 2007.

[40] M. Griebel, T. Dornseifer, T. Neunhoeffer, Numerical Simulation in Fluid Dynamics: A Practical Introduction, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[41] J. C. Tannehill, D. A. Anderson, R. H. Pletcher, Computational Fluid Mechanics and Heat Transfer, 2nd Edition, Taylor & Francis, 1997.

[42] A. J. Chorin, Numerical solution of the Navier–Stokes equations, Math. Comput. 22 (1968) 745–762.

[43] D. A. Jacobsen, I. Senocak, A full-depth amalgamated parallel 3D geometric multigrid solver for GPU clusters, in: 49th AIAA Aerospace Science Meeting, 2011.

[44] U. Ghia, K. N. Ghia, C. Shin, High-Re solutions for incompressible flow using the Navier–Stokes equations and a multigrid method, Journal of Computational Physics 48 (1982) 387–411.

[45] D. C. Wan, B. S. V. Patnaik, G. W. Wei, A new benchmark quality solution for the buoyancy-driven cavity by discrete singular convolution, Numerical Heat Transfer, Part B: Fundamentals 40 (3) (2001) 199–228.

[46] I. Senocak, J. Thibault, M. Caylor, Rapid-response urban CFD simulations using a GPU computing paradigm on desktop supercomputers, in: Eighth Symposium on the Urban Environment, Phoenix, Arizona, 2009.

[47] I. Senocak, D. Jacobsen, in: Fifth International Symposium on Computational Wind Engineering, CWE2010, Chapel Hill, North Carolina, USA, 2010.

[48] D. R. Butenhof, Programming with POSIX Threads, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
URL `http://portal.acm.org/citation.cfm?id=263953`

[49] The OpenMP ARB, OpenMP: A Proposed Industry Standard API for Shared Memory Programming, 1997.

[50] W. Gropp, R. Thakur, E. Lusk, Using MPI-2: Advanced Features of the Message Passing Interface, MIT Press, 1999.
URL `http://portal.acm.org/citation.cfm?id=555150`

[51] HPCwire, Microway 9U compact GPU cluster with octoputer, `http://www.microway.com/tesla/clusters.html` (Nov. 2009).

[52] B. Goglin, High throughput intra-node MPI communication with Open-MX, in: PDP, 2009, pp. 173–180.

[53] W. Gropp, R. Thakur, Thread-safety in an MPI implementation: Requirements and analysis, Parallel Computing 33 (9) (2007) 595–604.

[54] V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, W. mei Hwu, GPU clusters for high-performance computing, in: Proceedings of the IEEE Workshop on Parallel Programming on Accelerator Clusters, 2009.

[55] D. A. Jacobsen, J. C. Thibault, I. Senocak, An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters, in: 48th AIAA Aerospace Science Meeting, 2010.

[56] U. Trottenberg, C. Oosterlee, A. Schüller, Multigrid, Elsevier, 2001.

A flow solver is parallelized with MPI-CUDA and MPI-OpenMP-CUDA implementations.> Weak and strong scaling analysis performed using up to 256 GPUs> Three strategies to overlap computation and communication are assessed.> MPI-CUDA implementation with maximum overlapping gives the best performance> Tri-level parallelism does not show any advantage for the present application.