1-4-2011

Scalability of Incompressible Flow Computations on Multi-GPU Clusters Using Dual-Level and Tri-Level Parallelism

Dana A. Jacobsen
Boise State University

Inanc Senocak
Boise State University

49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition
4 - 7 January 2011, Orlando, Florida

AIAA 2011-947

Scalability of Incompressible Flow Computations on Multi-GPU Clusters Using Dual-Level and Tri-Level Parallelism

Dana A. Jacobsen*, and Inanc Senocak†
Boise State University, Boise, Idaho, 83725

High performance computing using graphics processing units (GPUs) is gaining popularity in the scientific computing field, with many large compute clusters being augmented with multiple GPUs in each node. We investigate hybrid tri-level (MPI-OpenMP-CUDA) parallel implementations to explore the efficiency and scalability of incompressible flow computations on GPU clusters up to 128 GPUS. This work details some of the unique issues faced when merging fine-grain parallelism on the GPU using CUDA with coarse-grain parallelism using OpenMP for intra-node and MPI for inter-node communication. Comparisons between the tri-level MPI-OpenMP-CUDA and dual-level MPI-CUDA implementations are shown using computationally large computational fluid dynamics (CFD) simulations. Our results demonstrate that a tri-level parallel implementation does not provide a significant advantage in performance over the dual-level implementation, however further research is needed to justify our conclusion for a cluster with a high GPU per node density or when using software that can utilize OpenMP's fine-grain parallelism more effectively.

I. Introduction

Graphics processing units (GPUs) have enjoyed rapid adoption within the high-performance computing (HPC) community. GPU clusters, where fast network connected compute-nodes are augmented with latest GPUs,[1] are now being used to solve challenging problems from various domains. To be specific, we define *multi-GPU clusters* as those where each compute-node of the cluster has at least two GPUs. Examples include the Lincoln Tesla cluster operated by the National Center for Supercomputing Applications (NCSA) at University of Illinois at Urbana Champaign[2] and the recent release of IBM's iDataPlex dx360 M3 mainstream HPC system using NVIDIA Tesla GPUs.[3]

Many multi-GPU clusters such as the previous examples use two GPUs per node. On these systems it can be efficient to use a dual-level parallel method using NVIDIA's Compute Unified Device Architecture (CUDA),[4] or Open Computing Language (OpenCL)[5] for fine-grain GPU parallellism and the Message-Passing Interface (MPI)[6] for coarse-grain parallelism. The overhead of inter-node communication between the two GPUs is generally not a first order effect. Some systems have densities as high as eight GPUs per node,[7,8] wherein the inter-node overhead of MPI can be substantial.

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 been well documented.[10–16]

Thibault and Senocak[15] developed a single-node multi-GPU 3D incompressible Navier-Stokes solver with a Pthreads-CUDA implementation that targets multi-GPU desktop platforms. This work was extended in Jacobsen et al.[16] where an MPI-CUDA implementation was presented and assessed on the NCSA Lincoln Tesla Cluster. These papers give details on the software implementation for multi-GPU clusters using a dual-level MPI-CUDA method. The present work builds upon these efforts and incorporates a third level of intra-node parallelism using OpenMP to arrive at a hybrid MPI-OpenMP-CUDA approach.

Cappello, Olivier, and Etiemble[17–19] 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 code by using a mixture of shared memory and message passing models. A number of other papers followed with the same conclusions[20–27]. Many of these papers also point out a number of cases where the applications

*Graduate Research Assistant, Department of Computer Science, Student Member AIAA.
†Assistant Professor, Department of Mechanical & Biomedical Engineering, Senior Member AIAA.

or computing systems are a poor fit to the hybrid model, and in some cases performance decreases. Lusk and Chan[28] 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 believe this combination of programming models is well fitted to modern scalable high performance systems.

Hager, Jost, and Rabenseifner[29] 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.[30] discuss issues arising from using MPI on petascale machines with close to a million processors. A number of MPI collective operations are shown to have exponential time with respect to the number of processors. 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 the authors to suggest a hybrid threading / MPI model as one way to mitigate the issue. However, in the case of a typical GPU system the situation is not as bad. In this case 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 indicates that clusters enhanced with GPUs look well suited for petascale and emerging exascale architectures. On the other hand, it also indicates that the hybrid model has less potential benefit on multi-GPU clusters.

Nakajima[31] 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 the present approach for the multi-GPU cluster implementation. 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[32].

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 they 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 numerous differences from dense-core CPU clusters. In this paper we investigate several methods of distributing computation using a tri-level parallel approach, using MPI for coarse-grain inter-node communication, OpenMP for medium-grain intra-node communication, and CUDA for fine-grain parallelism within the GPUs. We extend a dual-level MPI-CUDA 3D incompressible Navier-Stokes solver to use a hybrid MPI-OpenMP-CUDA approach.

II. Governing Equations and Numerical Approach

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

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

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

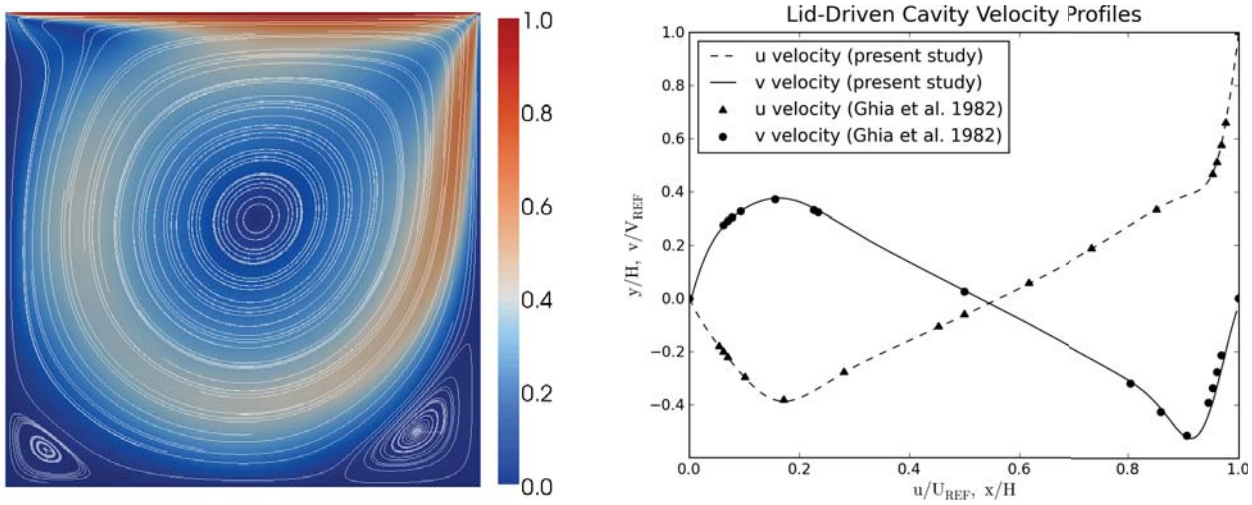
where $\mathbf{u}$ is the velocity vector, P is the pressure, ρ is the density, ν is the kinematic viscosity, and $\mathbf{f}$ is the body force. The Boussinesq approximation, which applies to incompressible flows with small temperature variations, is used to model the buoyancy effects in the momentum equations[33]:

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

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[34,35]

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



a) b)

Figure 1. Lid-driven cavity simulation with $\mathrm{Re} = 1000$ 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.[41].

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

A. Projection Algorithm

The buoyancy-driven incompressible form of the Navier-Stokes equations (Eqs. (1)-(4)) do not have and explicit equation for pressure. Therefore, a variety of methods have been proposed for splitting the solution into fractional steps where the momentum and pressure are independently solved. These include the projection algorithm of Chorin[36], Patankar's SIMPLE scheme[37,38] and its variants, and others. Many of these fractional-step methods are reviewed and contrasted in the survey of Guermond et al.[39].

The approach used in this study is the projection algorithm of Chorin, where the velocity field is predicted using the momentum equations without the pressure gradient term[36,40]. The resulting predicted velocity field does not satisfy the divergence free condition. By enforcing the divergence free condition on the velocity field at time $t+1$, a pressure Poisson equation can be derived from the momentum equations given in Eq. (2). The above equations are discretized 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 is solved using either a fixed iteration Jacobi solver or a geometric multigrid solver.

Validation on a number of test cases including the well-known lid-driven cavity and natural convection in heated cavity problems[41,42] 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 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.[41]. The results agree well with the benchmark data. The numerical results for the tri-level and dual-level parallel versions do not differ.

III. Multi-GPU Cluster Implementation of a 3D Incompressible Navier-Stokes Solver

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

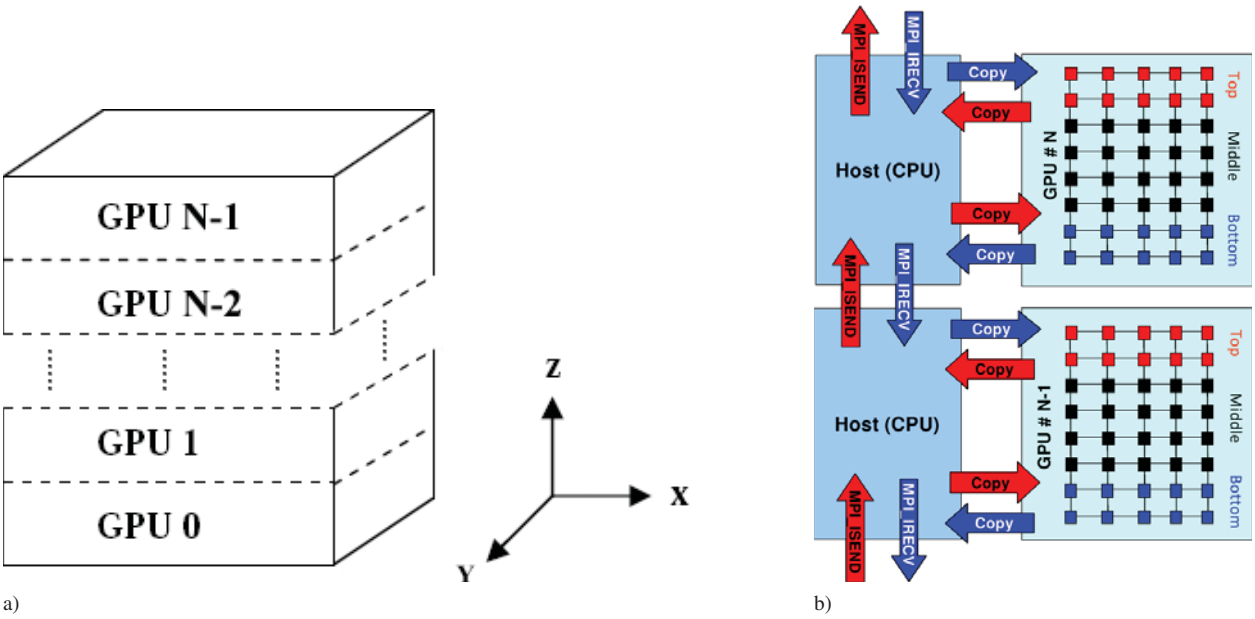


Figure 2. The domain decomposition. a) The decomposition of the full domain to the individual GPUs. b) An overview of the communication, GPU memory transfers, and the intra-GPU 1D decomposition used for overlapping.

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.

CUDA is the API used by NVIDIA for their GPUs.[4] CUDA programming consists of kernels that run on the GPU and are executed by all the processor units in a SIMD (Single Instruction Multiple Data) fashion. The CUDA API also extends the host C API with operations such as `cudaMemcpy()` which performs host/device memory transfers. Memory transfers between GPUs on a single host are done by using the host as an intermediary – there are no CUDA commands to operate between GPUs. On a given thread, CUDA kernel calls are asynchronous (i.e. control is given back to the host CPU before the kernel completes) but do not overlap (i.e. only one kernel runs at a time). Memory operations are synchronous and do not start until previous kernels have completed unless the CUDA streams functionality is used, which provides a mechanism for memory operations to run concurrently with kernel execution as well as host computation.

POSIX Threads[43] (Pthreads) and OpenMP[44] are two APIs used for running parallel code on a single machine using shared memory, such as widely available symmetric multiprocessor machines. These APIs both use a shared memory space model. Combined with CUDA, multiple GPUs on a single computer can perform computation, copy their neighboring cells to the host, synchronize with their neighbor threads, and copy the received boundary cells to the GPU for use in the next computational step.

The Message Passing Interface (MPI) API is widely used for parallel programming on clusters. MPI works on both shared and distributed memory machines. In general it will have some performance loss compared to the shared memory model used by threading APIs such as OpenMP and Pthreads, but in return it offers a highly portable solution to writing programs to work on a wide variety of machines and hardware topologies. Using MPI with one process mapped to each GPU is the most straightforward way to use a multi-GPU cluster.

A. Domain Decomposition

A 3D Cartesian volume is decomposed into 1D layers. These layers are then partitioned among the GPUs on the cluster to form a 1D domain decomposition. The 1D decomposition is shown in Fig. (2a). 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. CUDA is used within the GPUs to perform an orthogonal 2D decomposition of the data within each GPU.

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. 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 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 this reason, the 1D decomposition is 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 (2b) indicates how the 1D layers within each GPU are split into a top, bottom, and middle section. It also shows how communication and computation can be executed separately, allowing overlap. In an MPI-CUDA implementation, each process handles CUDA control and memory accesses for a single GPU, and multiple GPUs on a single compute-node can be managed by making multiple processes per node, and performing MPI transactions between each process regardless of whether it is on the same node. In contrast, the hybrid MPI-OpenMP-CUDA implementations create one process per compute-node, and an OpenMP thread per GPU. One or more additional threads may be used for MPI communication with the neighboring compute-nodes. For ghost cells interior to the compute-nodes, only OpenMP synchronization is necessary.

B. Implementation of the Projection Algorithm

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

    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 outer loop is used for time stepping, and indicates where the time step size can be adjusted. The `EXCHANGE` step updates the ghost cells for each GPU with the contents of the data from the neighboring GPU.

The projection algorithm described in section A. is implemented with pseudocode shown in listing 1. At each timestep there is also optional progress status output, and VTK visualization output. When the timestep loop ends,

the process writes the final output and clears GPU memory, then returns to the common code, which can exit. This method is used for both dual-level and tri-level implementations.

C. Tri-Level MPI-OpenMP-CUDA Implementations

To investigate whether additional efficiency can be gained from removing redundant message passing when processes are on the same host, a threading model is added. 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 this study.

- *MPI implementation efficiency*: The OpenMPI 1.3.2 software on the NCSA Lincoln Tesla cluster seems reasonably well optimized. Goglin[45] 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, as well as the extensive optimizations performed in Open-MX. Since the application being studied only using 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[19,29,31]. 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[19,21,23,28].

- *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. OpenMP has become the dominant method used in the HPC community, and it was decided this was the model to be used for this study. It is not believed that this choice had a noticeable performance impact, and OpenMP is clearer to read. The thread level parallelism is on a coarse grain level, since CUDA is handling the fine grain parallelism.

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.

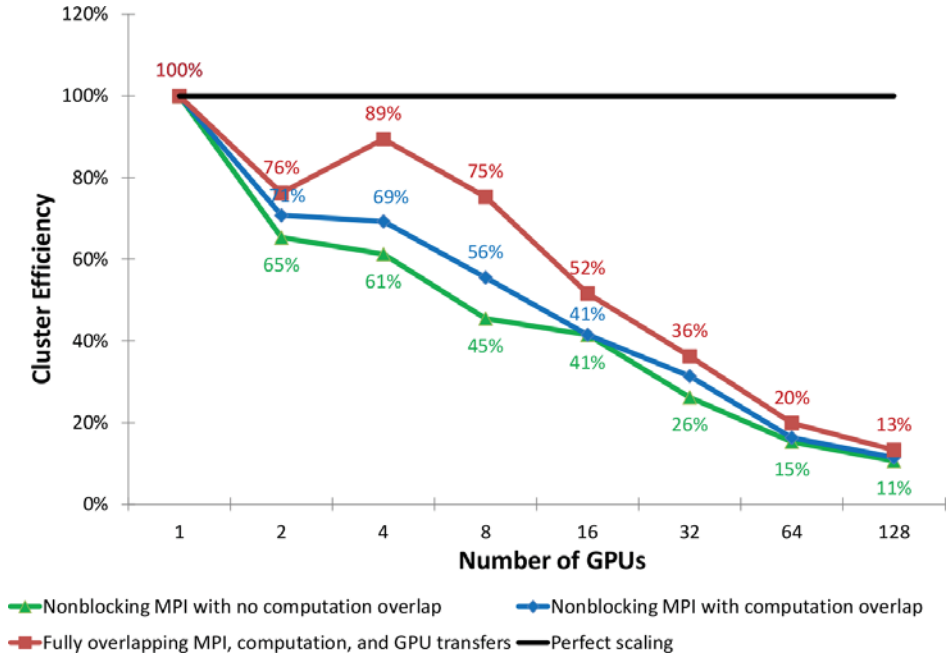
With many clusters having pre-installed versions of MPI libraries, sometimes with custom network infrastructure, it is not always possible to have access to the highest (`MULTIPLE`) threading level. Additionally, this level of threading support typically comes with some performance loss, so lower levels are preferred if they do not otherwise hinder parallelism[46]. 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.

IV. Performance Results from NCSA Lincoln and TACC Longhorn Clusters

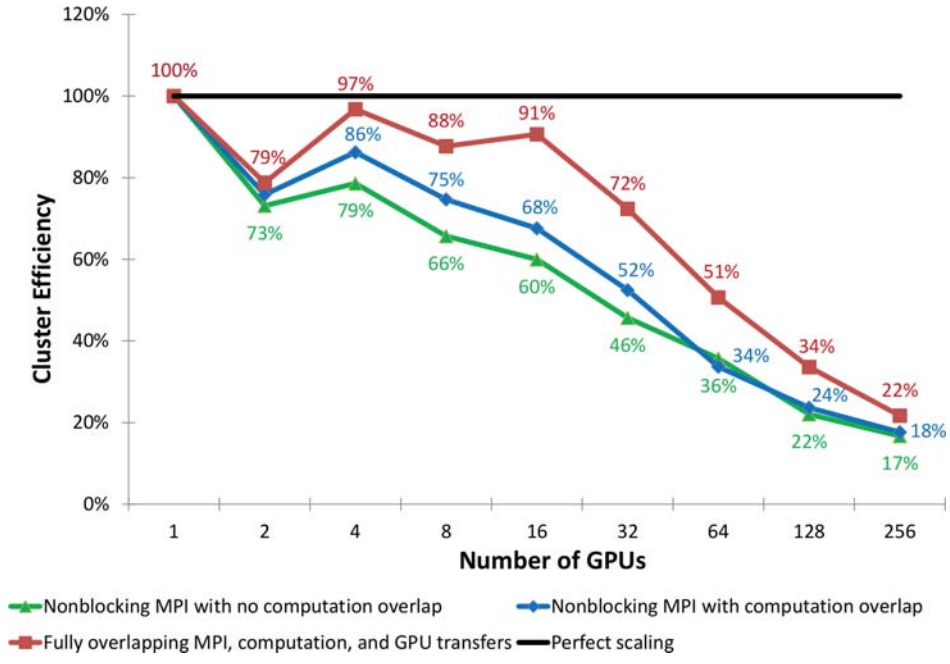
The NCSA Lincoln cluster consists of 192 Dell PowerEdge 1950 III servers connected via InfiniBand SDR (single data rate).[47] Each compute node has two quad-core 2.33GHz Intel64 processors and 16GB of host memory. The cluster has 96 NVIDIA Tesla S1070 accelerator units each housing four C1060-equivalent 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. In this study

```
// 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 2. An example Jacobi pressure loop using tri-level MPI-OpenMP-CUDA and simple computational overlapping. This uses the `SINGLE` threading level.



a) 3D Growth (Lincoln)



b) 3D Growth (Longhorn)

Figure 3. Efficiency of the three MPI-CUDA implementations with increasing number of GPUs (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) NCSA Lincoln Tesla cluster, b) TACC Longhorn cluster.

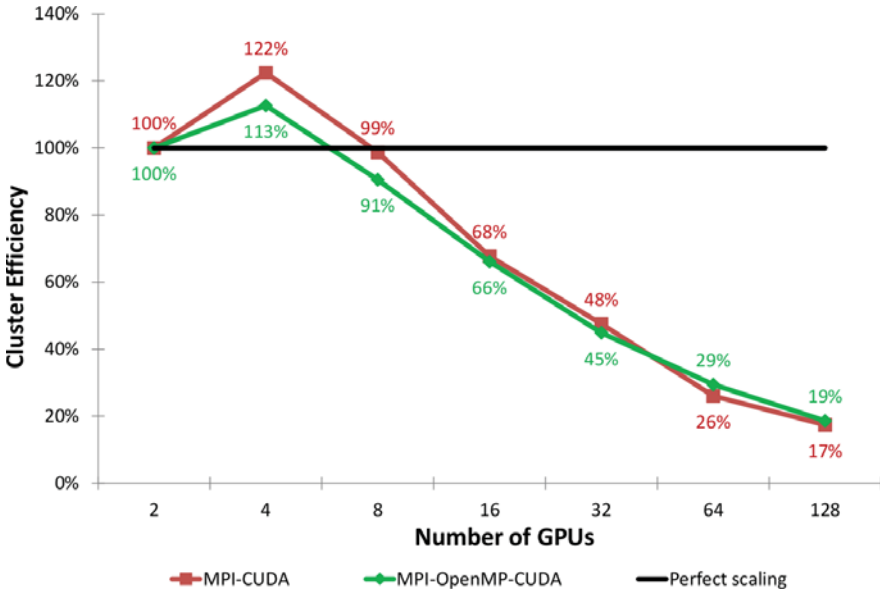


Figure 4. A comparison of weak scaling with the fully overlapped MPI-CUDA and single threaded MPI-OpenMP-CUDA implementations, with growth in three dimensions. Since the hybrid implementations use all the GPUs of a single node, the base value for parallel scaling is set to a single node of the NCSA Lincoln Tesla cluster containing two GPUs.

we show performance measurements for 64 of the 192 available compute-nodes in the Lincoln Tesla cluster, with 128 GPUs being utilized.

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 earlier, software issues on the NCSA Lincoln cluster precluded effective testing of anything but the tri-level implementation using single threading. Strong scaling and weak scaling measurements were performed, with little difference seen in most results. The weak scaling results with growth in three dimensions is the worst case for this application, and shows the most difference between the parallel methods. Figure (4) shows the the scaling efficiency of the fully overlapped dual-level MPI-CUDA and the tri-level MPI-OpenMP-CUDA implementations in the 3D growth weak scaling scenario. The MPI-CUDA data matches the fully overlapped data from Fig. (3), though 100% is set with two GPUs (a single node) rather than one.

With fewer than 4 nodes (8 GPUs), the dual-level MPI-CUDA implementation performs better. This may be due to the more inefficient synchronization methods used in the tri-level method with single-threaded MPI. With 32 and 64 nodes (64 and 128 GPUs), there is a small benefit with the 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. These results are consistent with the hybrid performance results shown by Nakajima[31], where MPI-vector outperformed his hybrid MPI-OpenMP-vector model at 64 and fewer nodes, and started showing an increasing benefit at 96 nodes and and beyond. We did not measure the results beyond this number of GPUs, but we believe the performance of the tri-level implementation should be further investigated on larger clusters.

V. Conclusions

We present a tri-level parallel implementation of the Navier-Stokes equations to simulate buoyancy-driven incompressible fluid flows on multi-GPU clusters with heterogeneous architectures. We adopt NVIDIA's CUDA programming model for fine-grain data-parallel operations within each GPU, OpenMP for parallel operations within individual compute-nodes, and MPI for coarse-grain parallelization across the cluster. We investigate the performance and scalability of incompressible flow computations on the NCSA Lincoln Tesla cluster and compare to previous dual-level parallel MPI-CUDA implementations[16].

A number of issues with obtaining the most benefit from tri-level MPI-OpenMP-CUDA parallel methods have

been identified. Compared to early results, current MPI libraries have much better optimization for multiple processes per node. A number of the benefits ascribed to the hybrid programming model are obtained via OpenMP's much better fine-grain parallelism support, which is not used at all in this study, since 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. 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. 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.

Another issue encountered is MPI library threading support. The MPI libraries must support the degree of threading support necessary to achieve best performance. None of the available MPI libraries on the NSCA Lincoln or TACC Longhorn clusters supported the highest levels of threading support. Because of the additional networking features of these clusters, compiling an out-of-the-box MPI library will not achieve maximum performance. The reason full threading support is not compiled in on by default is that it adds an additional overhead to all MPI calls. Therefore any benefit from mixing MPI and OpenMP must outweigh the small loss on every MPI call. With only 2-3 threads per process and no fine-grain parallelism via OpenMP, this is unlikely to occur in this model of tri-level parallelism.

Our performance measurements indicated the dual-level parallel model was better for small numbers of nodes, but showed a very small gain for larger numbers (32 to 64 nodes, 64 to 128 GPUs). Because of limitations with the MPI library noted previously, the implementation used was not optimal. We believe the gain from the tri-level MPI-OpenMP-CUDA parallel method is unlikely to exceed the detremental additional software complexity with the simulation model shown. Models that use fine-grain parallelism outside of CUDA, or have high GPU density per node will see better results.

Acknowledgments

This work is partially funded by grants from NASA Idaho Space Grant Consortium and National Science Foundation (NSF) (Award #1043107). 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]Showerman, M., Enos, J., Pant, A., Kindratenko, V., Steffen, C., Pennington, R., and Hwu, W.-M., "QP: A Heterogeneous Multi-Accelerator Cluster," *Proceedings of the 10th LCD International Conference on High-Performance Clustered Computing*, Boulder, Colorado, March 10–12 2009.

[2]NCSA, "Intel 64 Tesla Linux Cluster Lincoln webpage," http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/, 2008.

[3]HPCwire, "IBM Brings NVIDIA Tesla GPUs Onboard," http://www.hpcwire.com/features/IBM-Brings-NVIDIA-GPUs-Onboard-94190024.html, May 2010.

[4]NVIDIA, *NVIDIA CUDA Programming Guide 3.1.1*, 2010.

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

[6]Hempel, R., "The MPI Standard for Message Passing," *High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1994, Munich, Germany, April 18-20, 1994, Proceedings, Volume II: Networking and Tools*, edited by W. Gentzsch and U. Harms, Vol. 797 of *Lecture Notes in Computer Science*, Springer, 1994, pp. 247–252.

[7]HPCwire, "Colfax Unveils First Eight Tesla GPU Server," http://www.hpcwire.com/offthewire/Colfax-Server-Features-Eight-NVIDIA-Tesla-GPUs-64090602.html, Oct. 2009.

[8]HPCwire, "Microway 9U Compact GPU Cluster with OctoPuter," http://www.microway.com/tesla/clusters.html, Nov. 2009.

[9]Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J., "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, Vol. 26, No. 1, 2007, pp. 80–113.

[10]Brandvik, T. and Pullan, G., "Acceleration of a 3D Euler Solver using Commodity Graphics Hardware," *46th AIAA Aerospace Sciences Meeting and Exhibit*, Jan. 2008.

[11]Elsen, E., LeGresley, P., and Darve, E., "Large calculation of the flow over a hypersonic vehicle using a GPU," *Journal of Computational Physics*, Vol. 227, No. 24, Dec. 2008, pp. 10148–10161.

[12]Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Wobker, H., Becker, C., and Turek, S., "Using GPUs to Improve Multigrid Solver Performance on a Cluster," *International Journal of Computational Science and Engineering (IJCSE)*, Vol. 4, No. 1, 2008, pp. 36–55.

[13]Cohen, J. M. and Molemaker, M. J., "A Fast Double Precision CFD Code using CUDA," *Proceedings of Parallel CFD*, 2009.

[14]Schive, H., Tsai, Y., and Chiueh, T., "GAMER: a GPU-Accelerated Adaptive Mesh Refinement Code for Astrophysics," *Astrophysical Journal Supplement Series*, Vol. 186, 2010, pp. 457–484.

[15]Thibault, J. C. and Senocak, I., "CUDA Implementation of a Navier–Stokes Solver on Multi-GPU Platforms for Incompressible Flows," *47th AIAA Aerospace Science Meeting*, Jan. 2009.

[16]Jacobsen, D. A., Thibault, J. C., and Senocak, I., "An MPI-CUDA Implementation for Massively Parallel Incompressible Flow Computations on Multi-GPU Clusters," *48th AIAA Aerospace Science Meeting*, Jan. 2010.

[17]Cappello, F., Richard, O., and Etiemble, D., "Performance of the NAS Benchmarks on a Cluster of SMP PCs Using a Parallelization of the MPI Programs with OpenMP," *PaCT*, 1999, pp. 339–350.

[18]Cappello, F., Richard, O., and Etiemble, D., "Investigating the Performance of Two Programming Models for Clusters of SMP PCs," *HPCA*, 2000, pp. 349–359.

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

[20]Bova, S. W., Breshears, C. P., Cuicchi, C. E., Demirbilek, Z., and Gabb, H. A., "Dual-Level Parallel Analysis of Harbor Wave Response Using MPI and OpenMP," *Int. J. High Perform. Comput. Appl.*, Vol. 14, No. 1, 2000, pp. 49–64.

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

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

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

[24]Nakajima, K. and Okuda, H., "Parallel Iterative Solvers for Unstructured Grids Using an OpenMP/MPI Hybrid Programming Model for the GeoFEM Platform on SMP Cluster Architectures," *Proceedings of the 4th International Symposium on High Performance Computing*, Springer-Verlag, Kansai Science City, Japan, 2002, pp. 437–448.

[25]Rabenseifner, R., "Communication Bandwidth of Parallel Programming Models on Hybrid Architectures," *Proceedings of the 4th International Symposium on High Performance Computing*, Springer-Verlag, Kansai Science City, Japan, 2002, pp. 401–412.

[26]Rabenseifner, R., "Hybrid Parallel Programming on HPC Platforms," *EWOMP '03: Proceedings of the Fifth European Workshop on OpenMP*, Aachen, Germany, 2003, pp. 185–194.

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

[28]Lusk, E. and Chan, A., "Early Experiments with the OpenMP/MPI Hybrid Programming Model," *Lecture Notes in Computer Science*, Vol. 5004, 2008, pp. 36.

[29]Hager, G., Jost, G., and Rabenseifner, R., "Communication Characteristics and Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-core SMP Nodes," *Proceedings of the Cray Users Group Conference*, May 2009.

[30]Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Kumar, S., Lusk, E. L., Thakur, R., and Träff, J. L., "MPI on a Million Processors," *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.

[31]Nakajima, K., "Three-level hybrid vs. flat MPI on the Earth Simulator: Parallel iterative solvers for finite-element method," *Applied Numerical Mathematics*, Vol. 54, No. 2, 2005, pp. 237–255.

[32]Simon, H., Zacharia, T., and Stevens, R., "Modeling and Simulation at the Exascale for Energy and the Environment," Tech. rep., DOE ASCR Program, 2008.

[33]Kundu, P. K. and Cohen, I. M., *Fluid Mechanics*, Academic Press, 4th ed., 2007.

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

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

[36]Chorin, A. J., "Numerical Solution of the Navier–Stokes Equations," *Math. Comput.*, Vol. 22, 1968, pp. 745–762.

[37]Patankar, S. V. and Spalding, D. B., "A Calculation Procedure for Heat, Mass and Momentum Transfer in Three-Dimensional Parabolic Flows," *Int. J. Heat Mass Transfer*, Vol. 15, 1972, pp. 1787.

[38]Patankar, S. V., *Numerical Heat Transfer and Fluid Flow*, Taylor & Francis, 1980.

[39]Guermond, J. L. L., Minev, P., and Shen, J., "An overview of projection methods for incompressible flows," *Comput. Methods Appl. Mech. Engrg*, Vol. 196, 2006, pp. 6011–6045.

[40]Ferziger, J. H. and Perić, M., *Computational Methods for Fluid Dynamics*, Springer, 3rd ed., 2002.

[41]Ghia, U., Ghia, K. N., and Shin, C., "High-Re Solutions for Incompressible Flow Using the Navier–Stokes Equations and a Multigrid Method," *Journal of Computational Physics*, Vol. 48, 1982, pp. 387–411.

[42]Wan, D. C., Patnaik, B. S. V., and Wei, G. W., "A New Benchmark Quality Solution for the Buoyancy-Driven Cavity by Discrete Singular Convolution," *Numerical Heat Transfer, Part B: Fundamentals*, Vol. 40, No. 3, 2001, pp. 199–228.

[43]Butenhof, D. R., *Programming with POSIX Threads*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

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

[45]Goglin, B., "High Throughput Intra-Node MPI Communication with Open-MX," *PDP*, 2009, pp. 173–180.

[46]Gropp, W. and Thakur, R., "Thread-safety in an MPI implementation: Requirements and analysis," *Parallel Computing*, Vol. 33, No. 9, 2007, pp. 595–604.

[47]Kindratenko, V., Enos, J. J., Shi, G., Showerman, M. T., Arnold, G. W., Stone, J. E., Phillips, J. C., and mei Hwu, W., "GPU Clusters for High-Performance Computing," *Proceedings of the IEEE Workshop on Parallel Programming on Accelerator Clusters*, Aug. 2009.