2011 IEEE International Parallel & Distributed Processing Symposium

Overlapping Computation and Communication for Advection on Hybrid Parallel Computers

JB White III
Climate and Global Dynamics
National Center for Atmospheric Research
Boulder, Colorado
trey@ucar.edu

JJ Dongarra
Department of Electrical Engineering and Computer Science
University of Tennessee
Knoxville, Tennessee
dongarra@cs.utk.edu

***Abstract*—We describe computational experiments exploring the performance improvements from overlapping computation and communication on hybrid parallel computers. Our test case is explicit time integration of linear advection with constant uniform velocity in a three-dimensional periodic domain. The test systems include a Cray XT5, a Cray XE6, and two multicore Infiniband clusters with different generations of NVIDIA graphics processing units (GPUs). We describe results for Fortran implementations using various combinations of MPI, OpenMP, and CUDA, with and without overlap of computation and communication. We find that overlapping CPU computation, GPU computation, parallel communication, and CPU-GPU communication can provide performance improvements of more than a factor of two.**

***Keywords*-GPU; CUDA Fortran; MPI; OpenMP; linear advection;**

I. Introduction

Overlapping computation with communication has long been a strategy for improving throughput of parallel programs [1]–[5]. The idea is for parallel processes to perform independent computational work while the communication infrastructure performs message processing and data transfers, with the goal of hiding the latency and transfer costs of the inter-process communication.

In addition to multi-core processors (CPUs), parallel computers are starting to incorporate graphics processing units (GPUs) for double-precision floating-point computation. The GPUs typically have separate memory, and memory transfers between CPUs and GPUs typically occur over a bus, such as PCIe, with significant latency and transfer costs. The overlap technique used for parallel computation may also help for computations using GPUs. We describe experiments measuring the relative performance payoff for overlapping CPU computation, GPU computation, parallel communication, and CPU-GPU communication.

Our test case is explicit time integration of linear advection with constant uniform velocity in a three-dimensional periodic domain. We describe this case in more detail in Section II. Our target computers include a Cray XT5, a Cray XE6, and two multicore Infiniband clusters with different GPU generations; we provide details for these computers in Section III.

We test a variety of implementations of the test case, all in Fortran with OpenMP shared-memory parallelism:

- single task,
- bulk-synchronous MPI,
- MPI using nonblocking communication for overlap,
- MPI using OpenMP threading for overlap,
- GPU resident,
- GPU with bulk-synchronous MPI,
- GPU with MPI overlap using CUDA streams,
- GPU and CPU computation with bulk-synchronous MPI, and
- GPU and CPU computation partitioned for overlap with nonblocking MPI and CPU-GPU communication.

We describe these implementations in more detail in Section IV. Our implementations build off the work in [6], where that author investigates GPU implementations of a three-dimensional finite-difference computation, including implementations with MPI parallelism. We benefit from the algorithms described in [6], and we extend them with three-dimensional data decomposition for greater scalability, along with further decomposition to include the CPUs in the computation.

We present performance results in Section V. First we investigate the potential for MPI overlap with computation and the relative performance of different numbers of OpenMP threads per MPI task for a given total core count. We then explore the dependence of GPU performance on block size and the potential for overlap of GPU computation with CPU computation, MPI communication, and CPU-GPU communication. Finally we consider the effect of the CPU-GPU load balance.

Our performance results summarize a suite of runs for each implementation that spans the space of various tuning parameters, with particular emphasis on 1) the number of OpenMP threads per MPI task and 2) the relative size of the CPU and GPU computational domains. We do not perform automatic tuning, but we hope our results will inform efforts in automatic tuning, such as [7]–[9]. These works concen-

1530-2075 2011
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/IPDPS.2011.16

trate on automatically tuning multicore and GPU block sizes, but they do not address distributed-memory tuning or CPU-GPU load balancing.

We conclude in Section VI with a summary of our most-significant results and a discussion of their implications for automatic tuning and future architectures.

II. Test Case

Our long-term goal is to accelerate the simulation of climate and weather, and a prominent component of atmospheric dynamics is advection. Perhaps the simplest case of advection is linear advection with uniform constant velocity.

$$\partial_t u + \mathbf{c} \cdot \nabla u = 0 \qquad (1)$$

The symbol ∂_t indicates a partial derivative with respect to time. For three space dimensions, we have state $u = u(x, y, z, t)$ and uniform constant velocity $\mathbf{c} = \{c_x, c_y, c_z\}$.

Our test domain is a three-dimensional cube with periodic boundaries, and the initial condition for $u(t = 0)$ is a Gaussian wave at the center of the cube. Equation 1 moves the wave in the direction of the velocity without changing its shape.

We choose a strong-scaling problem for our test, where the global problem size stays the same as the number of parallel processes increases. Changing the grid size for climate simulations is typically a complex task because of various physical-process parameterizations that may depend on the grid size, so climate simulations are typically strong-scaling problems.

We discretize in space using a uniform grid of $420 \times 420 \times 420$ points, again with periodic boundaries. We choose this size to just fit within the memory of a single GPU, with each dimension divisible by the GPU "warp" size [10] plus the domain overlap. Section V-C has more information about GPU block sizes and their performance impact.

We discretize in time using an explicit Lax-Wendroff technique [11]; we expand the state $u(t + \Delta)$ in a Taylor series around small time step Δ and use Equation 1 to change time derivatives into space derivatives. We use a $3 \times 3 \times 3$ stencil centered around the point $u(x, y, z, t)$, with unknown coefficients, and expand in a Taylor series around the small grid spacing δ. We subtract the time expansion from the space expansion and form equations for the coefficients by canceling terms. With a $3 \times 3 \times 3$ stencil, we can cancel all terms through $O(\Delta^2)$ and some higher-order terms.

The resulting discrete equation for a single time step is the following.

$$u(x,y,z,t+\Delta) \approx \sum_{i,j,k=-1..+1} a_{ijk} u(x+i\delta, y+j\delta, z+k\delta, t) \qquad (2)$$

The values of a_{ijk} appear in Table I, in terms of the constant uniform velocity components c_x, c_y, and c_z and the ratio

a_{-1-1-1}	$c_x c_y c_y \nu^3 (1 + c_x\nu)(1 + c_y\nu)(1 + c_z\nu)/8$
$a_{-1-1\ 0}$	$-2c_x c_y \nu^2 (1 + c_x\nu)(1 + c_y\nu)(c_z^2\nu^2 - 1)/8$
a_{-1-1+1}	$c_x c_y c_z \nu^3 (1 + c_x\nu)(1 + c_y\nu)(c_z\nu - 1)/8$
$a_{-1\ 0-1}$	$-2c_x c_z \nu^2 (1 + c_x\nu)(1 + c_z\nu)(c_y^2\nu^2\text{-}1)/8$
$a_{-1\ 0\ 0}$	$4c_x\nu(1 + c_x\nu)(c_y^2\nu^2 - 1)(c_z^2\nu^2 - 1)/8$
$a_{-1\ 0+1}$	$-2c_x c_z \nu^2 (1 + c_x\nu)(-1 + c_z\nu)(-1 + c_y^2\nu^2)/8$
a_{-1+1-1}	$c_x c_y c_z \nu^3 (1 + c_x\nu)(-1 + c_y\nu)(1 + c_z\nu)/8$
$a_{-1+1\ 0}$	$-2c_x c_y \nu^2 (1 + c_x\nu)(-1 + c_y\nu)(-1 + c_z^2\nu^2)/8$
a_{-1+1+1}	$c_x c_y c_z \nu^3 (1 + c_x\nu)(-1 + c_y\nu)(-1 + c_z\nu)/8$
a_{0-1-1}	$-2c_y c_z \nu^2 (1 + c_y\nu)(1 + c_z\nu)(-1 + c_x^2\nu^2)/8$
$a_{0-1\ 0}$	$4c_y\nu(1 + c_y\nu)(-1 + c_x^2\nu^2)(-1 + c_z^2\nu^2)/8$
a_{0-1+1}	$-2c_y c_z \nu^2 (1 + c_y\nu)(-1 + c_z\nu)(-1 + c_x^2\nu^2)/8$
$a_{0\ 0-1}$	$4c_z\nu(1 + c_z\nu)(-1 + c_x^2\nu^2)(-1 + c_y^2\nu^2)/8$
$a_{0\ 0\ 0}$	$-8(-1 + c_x^2\nu^2)(-1 + c_y^2\nu^2)(-1 + c_z^2\nu^2)/8$
$a_{0\ 0+1}$	$4c_z\nu(-1 + c_z\nu)(-1 + c_x^2\nu^2)(-1 + c_y^2\nu^2)/8$
a_{0+1-1}	$-2c_y c_z \nu^2 (-1 + c_y\nu)(1 + c_z\nu)(-1 + c_x^2\nu^2)/8$
$a_{0+1\ 0}$	$4c_y\nu(-1 + c_y\nu)(-1 + c_x^2\nu^2)(-1 + c_z^2\nu^2)/8$
a_{0+1+1}	$-2c_y c_z \nu^2 (-1 + c_y\nu)(-1 + c_z\nu)(-1 + c_x^2\nu^2)/8$
a_{+1-1-1}	$c_x c_y c_z \nu^3 (-1 + c_x\nu)(1 + c_y\nu)(1 + c_z\nu)/8$
$a_{+1-1\ 0}$	$-2c_x c_y \nu^2 (-1 + c_x\nu)(1 + c_y\nu)(-1 + c_z^2\nu^2)/8$
a_{+1-1+1}	$c_x c_y c_z \nu^3 (-1 + c_x\nu)(1 + c_y\nu)(-1 + c_z\nu)/8$
$a_{+1\ 0-1}$	$-2c_x c_z \nu^2 (-1 + c_x\nu)(1 + c_z\nu)(-1 + c_y^2\nu^2)/8$
$a_{+1\ 0\ 0}$	$4c_x\nu(-1 + c_x\nu)(-1 + c_y^2\nu^2)(-1 + c_z^2\nu^2)/8$
$a_{+1\ 0+1}$	$-2c_x c_z \nu^2 (-1 + c_x\nu)(-1 + c_z\nu)(-1 + c_y^2\nu^2)/8$
a_{+1+1-1}	$c_x c_y c_z \nu^3 (-1 + c_x\nu)(-1 + c_y\nu)(1 + c_z\nu)/8$
$a_{+1+1\ 0}$	$-2c_x c_y \nu^2 (-1 + c_x\nu)(-1 + c_y\nu)(-1 + c_z^2\nu^2)/8$
a_{+1+1+1}	$c_x c_y c_z \nu^3 (-1 + c_x\nu)(-1 + c_y\nu)(-1 + c_z\nu)/8$

Table I
VALUES OF COEFFICIENTS a_{ijk} USED IN EQUATION 2, WHERE c_x, c_y, AND c_z ARE THE VELOCITY COMPONENTS AND $\nu = \Delta/\delta$.

$\nu = \Delta/\delta$. Note that the values of a_{ijk} for this test are the same for every grid point and time step because the velocity is constant and uniform. Our method is $O(\Delta^3)$ for a single time step and $O(\Delta^2)$ for a fixed simulated time. It is numerically stable for $\nu \leq \max\{|c_x|, |c_y|, |c_z|\}$, and we run the test at the maximum stable value of ν.

To measure performance, we measure the time to perform multiple time steps. We vary the number of steps to ensure that each experiment runs long enough for accurate measurements, at least 5 seconds per measurement. Given the measured time in seconds, the grid size, and the number of times steps, we analytically compute the performance in GF (billions of floating-point operations per second) based on the 53 floating-point operations appearing in Equation 2: 27 multiplications and 26 additions.

III. Computers

We present results for four computers: a Cray XT5, a Cray XE6, and two multicore Infiniband clusters with different generations of NVIDIA GPUs. Technical details of each computer are given in Table II. The Cray XT5 is JaguarPF, the primary computer at the Oak Ridge Leadership Computing Facility (OLCF), with a peak performance of 2.3 PF. The Cray XE6 is Hopper II, the primary computer at the National Energy Research Scientific Computing Center (NERSC), with a peak performance of almost 1.3 PF. Lens is the OLCF analysis cluster and includes GPUs originally intended to

System	JaguarPF	Hopper II	Lens	Yona
Compute nodes	18688	6392	31	16
Memory per node (GB)	16	32	64	32
AMD Opteron sockets per node	2	2	4	2
Cores per Opteron socket	6	12	4	6
Opteron clock (GHz)	2.6	2.1	2.3	2.6
Interconnect	Cray SeaStar 2+	Cray Gemini	DDR Infiniband	QDR Infiniband
MPI	Cray MPT 4.0.0	Cray MPT 5.1.3	OpenMPI 1.3.3	OpenMPI 1.7a1
NVIDIA Tesla GPU	–	–	C1060	C2050
GPU memory (GB)	–	–	4	3

Table II
TECHNICAL DETAILS OF TESTED COMPUTERS.

support visualization. Each node of Lens has an NVIDIA Tesla C1060 capable of fast double-precision floating-point operations. The final computer is Yona, an experimental OLCF cluster with newer, faster NVIDIA Tesla C2050 GPUs and a faster PCIe bus connecting the GPUs to the CPUs and main memory. The primary intent of the GPUs on Yona is for general-purpose computation, including double-precision floating-point computation. We use PGI Fortran 10.9 on Hopper II at NERSC and 10.6 on the OLCF computers. On Lens and Yona, the OLCF GPU clusters, this includes PGI CUDA Fortran. We compile all cases with the options "`-mp -fast -gopt -Minfo=all`". On Lens we add "`-Mcuda=cc13`", and on Yona we add "`-Mcuda=cc20`". These additional options reflect the most-recent CUDA versions supported by the respective GPUs.

IV. IMPLEMENTATIONS

We test a variety of implementations designed to measure the relative performance improvement from overlapping CPU computation, GPU computation, parallel communication, and CPU-GPU communication. Each implementation is Fortran with OpenMP directives, most include MPI for parallel communication, and the GPU implementations include CUDA Fortran. See the following subsections for details.

A. *Single Task*

The baseline implementation uses a single task with multiple threads. We use the Fortran intrinsic `system_clock` to measure the wall-clock time of the time steps. Each time step has three algorithmic steps.

1) Copy periodic boundaries.
2) Compute the new state using Equation 2.
3) Copy the new state to the current state.

Step 1 copies boundary points into halo, or ghost, points on the opposite boundary, and Step 2 uses the halo points. Step 1 uses doubly nested loops, and Steps 2 and 3 use triply nested loops. We use OpenMP to parallelize these loops, the outer loops in Step 1 and the outer-most two loops in Steps 2 and 3 (using the OpenMP option `collapse(2)`). We verify the implementation by recording norms of the difference between the computed state and the analytic state.

B. *Bulk-Synchronous MPI*

The bulk-synchronous MPI implementation adds distributed-memory parallelism to the single-task implementation. The data-distribution algorithm gives each task a subdomain that is as close to the same size as possible and as close to cubic as possible, with the constraint that no task gets an empty domain. If the number of tasks is the cube of an integer, and if that integer is a divisor of 420 (the domain size in each dimension), then every task has a cubic subdomain of the same size. In other cases, the subdomain size is largest in the x dimension and smallest in the z dimension, to best enable memory locality. The largest subdomain is at most one grid point larger in each dimension than the smallest. The subdomains are aligned in each dimension, so each MPI task has 26 neighbors. Note that a task may be its own neighbor in decompositions with small or prime numbers of tasks.

We perform a barrier immediately before measuring the start time and the end time. The implementation is bulk synchronous: it performs all of Step 1 from Section IV-A, through parallel communication, before proceeding to Steps 2 and 3, which involve only local computation. And those steps complete before starting Step 1 for the next iteration.

To perform Step 1, the master thread first issues nonblocking receive calls for 6 neighbors. Serially in each dimension, all threads copy into send buffers, the master thread sends and completes the receives, and all threads copy from receive buffers into halos. The dimensions are serialized so that the x corners can be sent to y neighbors, and x and y to z. This well-established strategy reduces the number of neighbor exchanges from 26 to 6 for this three-dimensional case.

C. *MPI Using Nonblocking Communication for Overlap*

We attempt to overlap computation and communication using a common strategy; we partition the computation in Step 2 of Section IV-A and interleave the partitioned computations with substeps of Step 1. We first partition each local domain into interior points and boundary points, where the boundary points are those that touch halo points. We then partition the interior points into thirds along the z dimension. The first third executes between nonblocking initiation of the x communication and its completion, the second third within the y communication, and the final third within the z. The threads compute the boundary points after the communication.

D. MPI Using OpenMP Threading for Overlap

Instead of using nonblocking MPI communication, we attempt to overlap computation and communication in this implementation using an asynchronous OpenMP thread to perform the MPI communication. We again partition the computation in Step 2 of Section IV-A into interior and boundary points. The master thread (`!$omp master`) performs the MPI communication and then joins in the computation of the interior points, while the other threads begin computation on the interior points immediately. We implement this by changing the scheduling of the threading for the interior points to `schedule(guided)`, which distributes chunks of work as threads request them, with chunks proportional in size to the remaining work divided by the number of threads [12]. An OpenMP barrier ensures that the master thread completes communication before computation begins on the boundary points.

E. GPU Resident

Recall that the problem size is roughly as large as possible while still fitting within the memory of a single GPU, called its "global memory". This constraint allows us to compare against the best-case scenario for GPU performance, where the problem resides within the GPU memory for the length of the computation, with no memory exchanges with the CPUs.

Our GPU implementation uses CUDA Fortran [10] and is based on the algorithm in [6]. The a_{ijk} values are in GPU "constant memory". We partition the domain along the x and y dimensions such that each two-dimensional thread block gets a unique xy block, along with a halo. The threads iterate over the z direction. On each iteration, a thread block copies a slab of xy points from global memory into "shared memory", memory that is local to the thread block and shared among the threads in the block. Note that the thread block includes threads associated with halo points that only perform memory operations. Halo threads beyond the boundary of the global domain copy from the opposite boundary to implement periodicity. The interior threads compute and update local state variables and store the completed computations in global memory.

The CPU issues a CUDA kernel call for each time step, flipping the arguments between two GPU state variables to avoid the need for an extra copy operation. The CPU and GPU synchronize immediately before timer calls. We do not include the time to copy the initial state to the GPU or copy the final result from it. This represents a best-case scenario for GPU computation, where a computation might run for hours between CPU-GPU checkpoints, and the relative cost of copying between CPU and GPU is negligible.

F. GPU with Bulk-Synchronous MPI

This multi-GPU implementation uses CPUs to perform MPI communication. We partition the domain among tasks as in the CPU-only MPI implementations. Instead of a single GPU kernel, we define separate kernels for the interior points and for each pair of boundary faces in each dimension. The kernel for the interior points is a simplified version of the single-GPU kernel, without the logic for copying opposite boundaries. Each boundary-face kernel copies halo values from a buffer and writes boundary values to both the state variable and an outgoing buffer. We need the buffers to allow communication between CPU and GPU to be in large contiguous chunks.

For each time step, a CPU copies boundary buffers from the GPU, communicates the boundaries as in the CPU-only bulk-synchronous implementation, copies halo buffers back to the GPU, and makes kernel calls for the faces and interior. Note that the target computers have more CPU cores than GPUs, and we can have more than one MPI task issuing calls to a particular GPU. The number of MPI tasks per GPU is a tunable performance parameter.

G. GPU with MPI Overlap Using CUDA Streams

This implementation uses multiple CUDA streams to overlap computation of interior points with CPU-GPU communication and MPI communication. For each time step, a CPU first issues a kernel call to one CUDA stream for the computation of interior points. It then performs the MPI communication and issues kernel calls to a second stream to copy halo buffers to the GPU, compute the boundary values, and copy the boundary buffers back from the GPU. The interior computation can thus overlap the MPI communication, buffer copies, and, on some GPUs, the boundary computation. The CPU ends the time step by synchronizing the two CUDA streams.

H. CPU and GPU Computation with Bulk-Synchronous MPI

This implementation computes on both the CPUs and GPUs. We partition each task's domain between CPU and GPU as a block in a box. The GPU is responsible for the interior block, and the CPU is responsible for an enclosing box. Figure 1 gives a two-dimensional representation of this decomposition. We can tune the thickness of the box walls to balance the load between the CPU and GPU.

A CPU task starts each time step by exchanging inner halo and boundary buffers with the GPU and outer halos and boundaries with other tasks through MPI. It then issues the GPU kernels for the inner block points and computes the outer box points. The CPU and GPU may thus overlap computation.

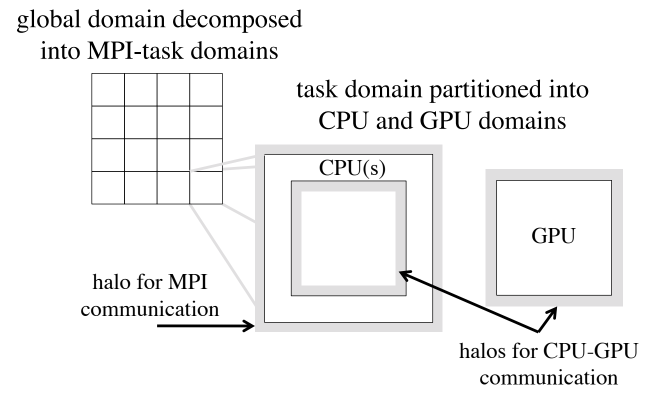
global domain decomposed into MPI-task domains
task domain partitioned into CPU and GPU domains
CPU(s)
GPU
halo for MPI communication
halos for CPU-GPU communication

Figure 1. Domain decomposition for CPU-GPU implementations described in Sections IV-H and IV-I. The test domain is three dimensional, but this figure is simplified to two dimensions.

I. *CPU and GPU Computation Partitioned for Overlap with Nonblocking MPI and CPU-GPU Communication*

This implementation computes on both CPUs and GPUs and attempts the most-extensive overlap. It uses the same CUDA kernels and domain decomposition (Figure 1) as the implementation in Section IV-H, but it uses separate CUDA streams for the GPU interior and boundary points. A CPU task starts each time step by issuing a kernel call for the GPU interior points. It then issues nonblocking MPI receives and asynchronous memory copies to the GPU, followed by kernel calls for the GPU boundary points and asynchronous memory copies from the GPU. It overlaps MPI communication in each dimension with the computation of CPU interior points of that same dimension. For example, it overlaps communication to the $\pm x$ neighbors with computation of the interior and inner-boundary points of the $\pm x$ walls of the box in Figure 1. Finally it computes the outer boundary points and synchronizes the CUDA streams.

This implementation has the potential to overlap CPU computation, GPU computation, MPI communication, and CPU-GPU communication. Because it may overlap more than two types of operation, this implementation may improve performance by more than a factor of two. The thickness of the CPU box domain is again a tunable parameter to balance the load between CPUs and GPUs.

These implementations vary greatly in complexity, and the number of lines of code can hint at the programmer-productivity costs of the various strategies for improving performance. Figure 2 shows the lines of Fortran code for each implementation, minus blank lines and lines containing only comments. MPI parallelization adds 57–73% more lines, with the nonblocking overlap adding the most. Targeting a single GPU with CUDA Fortran uses just 6% more lines than targeting a single process with OpenMP threading, but adding MPI parallelism to the GPU computation almost triples the number of lines. The combination of CPU computation, GPU computation, and MPI parallelism is most expensive, with the full-overlap implementation using exactly four times as many lines as the single-process multithreaded implementation (860 versus 215).

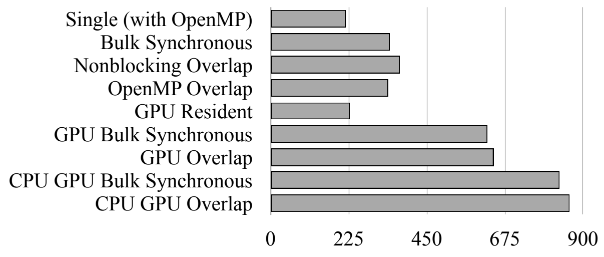

Figure 2. Lines of Fortran code for each implementation, minus blank lines and comments.

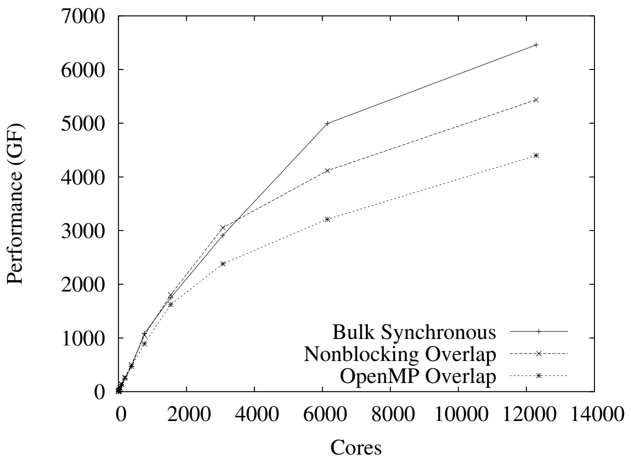

Figure 3. The best performance of each JaguarPF implementation for a range of core counts.

V. Results

A. *MPI Performance and Overlap*

First we consider the potential performance improvement of MPI overlap. Figure 3 shows the performance of each implementation on JaguarPF for a range of core counts. JaguarPF has no GPUs, so no GPU implementations are included. Each value is the best result for a given number of cores, among all measured numbers of OpenMP threads per MPI task. Because JaguarPF has two 6-core sockets per node, we include measurements for 1, 2, 3, 6, and 12 threads per task.

For core counts below 4000, the implementation with overlap from nonblocking communication (Section IV-C) can slightly outperform the bulk-synchronous implementation (Section IV-B). At 6000 and above, as the work

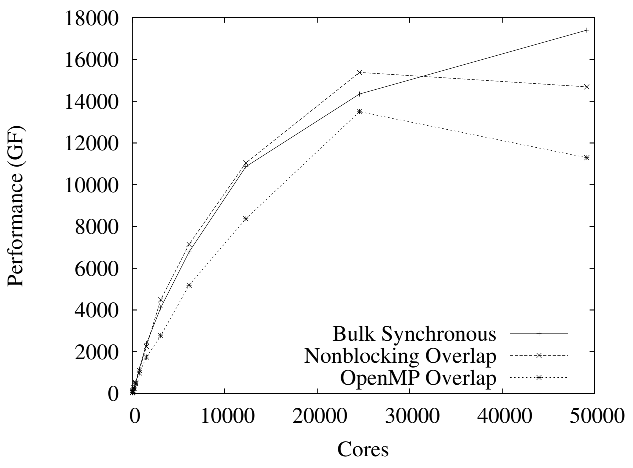

Figure 4. The best performance of each Hopper-II implementation for a range of core counts.

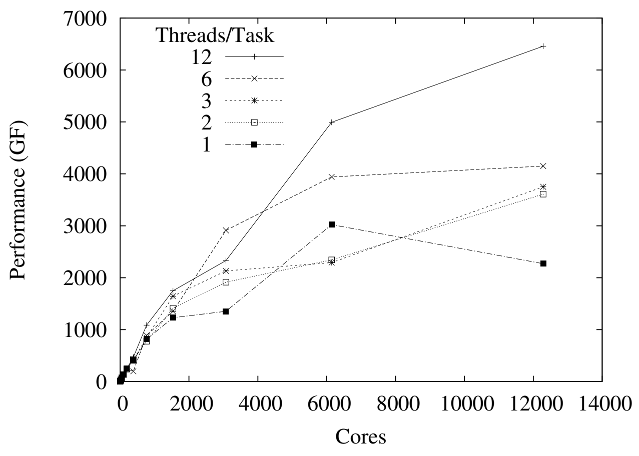

Figure 5. The performance of the bulk-synchronous implementation (Section IV-B) on JaguarPF for a range of core counts and various numbers of OpenMP threads per MPI task.

per core dwindles, the bulk-synchronous implementation has a significant advantage. The implementation using an OpenMP thread for overlap (Section IV-D) consistently lags in performance.

Figure 4 shows analogous results for Hopper II. It has two 12-core sockets per node, where each socket has two 6-core chips, so we include measurements for 1, 2, 3, 6, 12, and 24 threads per task. Likely because of the newer Gemini interconnect, Hopper II scales better than JaguarPF, so we include results out to 49152 cores. Like for JaguarPF, the implementation with overlap from nonblocking communication (Section IV-C) performs slightly better than the bulk-synchronous implementation (Section IV-B) for core counts below some limit, but that limit is an order of magnitude higher on Hopper II. Again the implementation using an OpenMP thread for overlap (Section IV-D) consistently lags in performance.

Figures 9 and 10, which we explain in detail in Section V-D, show results for Lens and Yona. For our test case on these smaller computers, overlap of computation and communication improves performance little or none at all.

B. OpenMP Threads Per MPI Task

Each result in Figures 3 and 4 is for the best-performing number of OpenMP threads per MPI task for that number of cores. Here we consider one implementation on JaguarPF and Hopper II, the bulk-synchronous one (Section IV-B), and explore the performance impact of the number of threads per task. Figures 5 and 6 show the results for JaguarPF and Hopper II, respectively.

We see that different numbers of threads per task perform best at different total core counts. It may not be clear from the figures, but each of 1, 2, 3, 6, and 12 threads per MPI task performs best for at least one of the tested number

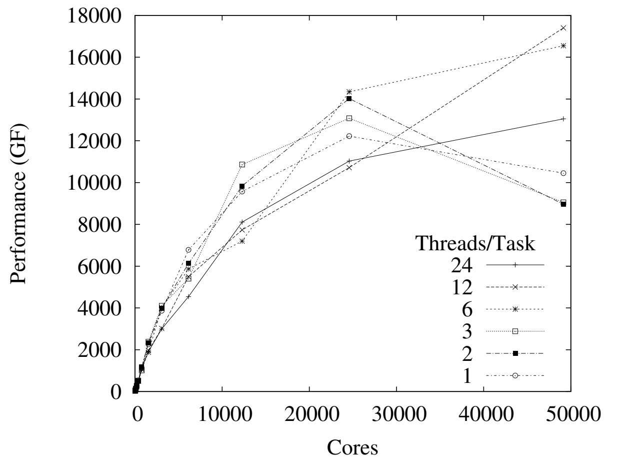

Figure 6. The performance of the bulk-synchronous implementation (Section IV-B) on Hopper II for a range of core counts and various numbers of OpenMP threads per MPI task.

of cores on each computer. Only 24 threads per task (on Hopper II) is never optimal. On JaguarPF the best number of threads per task generally increases as the total number of cores increases. The results vary more on Hopper II, but larger numbers of threads per task are best at the highest core counts. Unlike the case of communication overlap, our tests show significant performance improvement from hybrid MPI and OpenMP parallelism, particularly at high core counts.

To save space, we do not present figures analogous to Figures 5 and 6 for Lens and Yona. They also show significant variability in the best number of threads per task. On Lens, which has four 4-core sockets per node, we have measurements for 1, 2, 4, 8, and 16 threads per task. The best number for our test is either 4, 8, or 16, with no clear correlation with total core count. On Yona, which

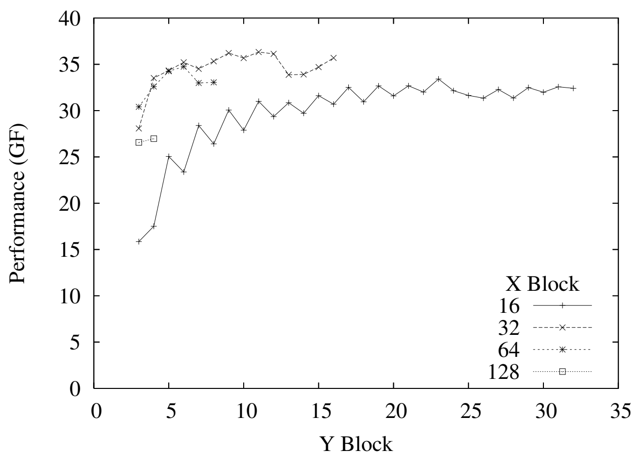

Figure 7. Performance of the GPU-resident implementation (Section IV-E) on Lens using a variety of two-dimensional block sizes.

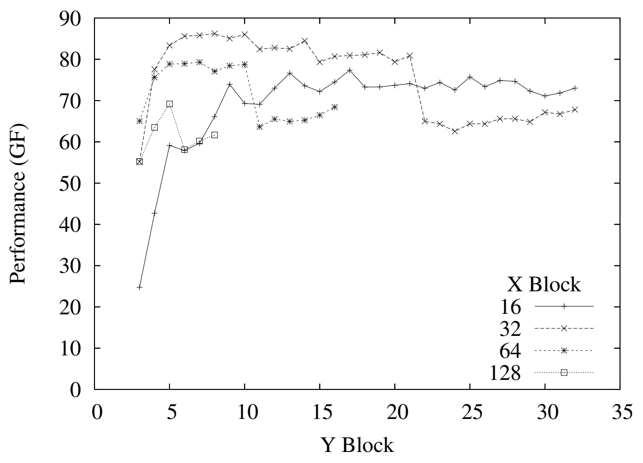

Figure 8. Performance of the GPU-resident implementation (Section IV-E) on Yona using a variety of two-dimensional block sizes.

like JaguarPF has two 6-core sockets per node, we have measurements for 1, 2, 3, 6, and 12 threads per task. For its relatively small core counts, the best number of threads per task is 1, 2, 3, or 6. Like JaguarPF, Yona shows a general increase in the best number of threads per task as the total core count increases.

C. GPU Block Size

Our GPU implementations have many performance variables, including: overlap strategy, OpenMP threads per MPI task (and thus MPI tasks per GPU), CPU-GPU load balance, and GPU block size. To simplify our analysis, we first consider block sizes for the GPU-resident implementation (Section IV-E).

Figure 7 shows the performance on Lens for a variety of two-dimensional block sizes. The C1060 GPUs on Lens support three-dimensional block sizes of up to 512 elements, and they have a "warp" size of 32. Memory access is fastest for contiguous blocks of at least a half warp, so we only consider x dimensions of 16, 32, 64, and 128. We use two-dimensional blocks instead of three because they allow better memory reuse in our test. We vary the y dimension up to the maximum total size of 512 elements.

An x dimension of 32, the warp size, tends to provide the best performance, with the top performance coming from a block size of 32×11. We use this block size for all our parallel GPU experiments on Lens. (See [7] for an investigation of automatic tuning of GPU block size.)

Figure 8 shows the analogous performance on Yona . The C2050 GPUs on Yona support block sizes of up to 1024 elements, and they have a "warp" size of 32. Again, the best performance comes from an x block size of 32, but with a slightly smaller y block size of 8. We use this block size, 32×8, for all our parallel GPU experiments on Yona.

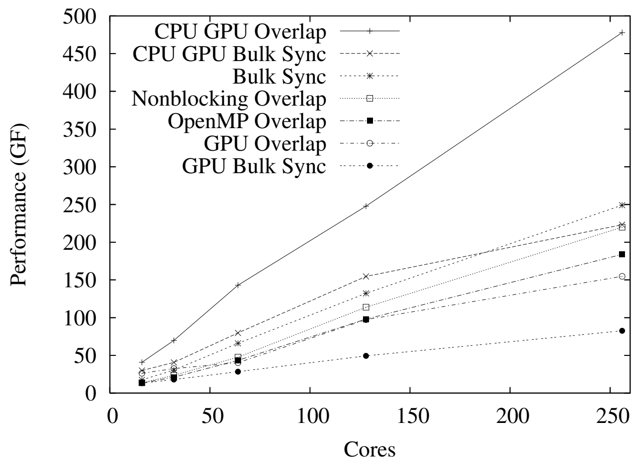

Figure 9. The best performance of each Lens implementation for a range of core counts. The GPU implementations use one GPU per 16 cores.

D. Parallel GPU Performance and Overlap

Figure 9 shows the performance of each implementation on Lens for a range of core counts. Each value is the best performance for that implementation, among a variety of threads per task and, where applicable, box thicknesses (from Figure 1). The CPU-only implementations benefit little from overlap, but the GPU implementations benefit greatly from overlap, particularly for the full-overlap case (Section IV-I), where CPU computation, GPU computation, MPI communication, and CPU-GPU communication can occur concurrently. In fact, the best CPU-GPU performance exceeds the sum of the best CPU-only performance plus the best GPU-computation performance.

The results for Yona are still more striking. Figure 10 shows the best performance of each implementation for a range of core counts. The GPUs are a larger fraction of the computational power on Yona than on Lens, so

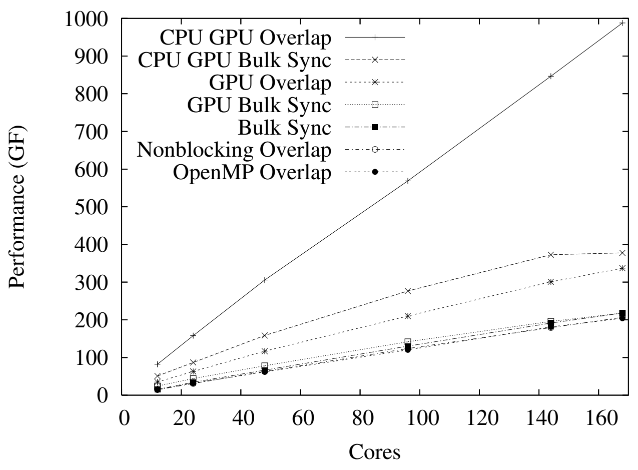

Figure 10. The best performance of each Yona implementation for a range of core counts. The GPU implementations use one GPU per 12 cores.

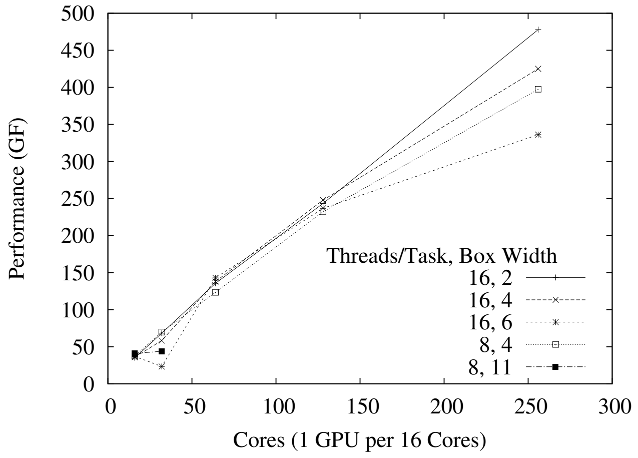

Figure 11. The performance of the CPU-GPU overlap implementation (Section IV-I) on Lens for various combinations of OpenMP threads per MPI task and box thickness.

the performance of the best CPU-GPU implementation is more than four times the performance of the best CPU-only implementation.

E. CPU-GPU Load Balancing and Overlap

Next we consider the performance of the CPU-GPU overlap implementation for different numbers of threads per task and different box thicknesses. Figure 11 shows this performance on Lens. Each combination plotted has the best performance for at least one core count. In general, the best performance comes from few tasks per node, and the best box width decreases with increasing core count. This decrease makes sense because the amount of work per core also decreases with core count.

Figure 12 shows the performance of the CPU-GPU overlap implementation on Yona. Again, each combination plotted has the best performance for at least one core count.

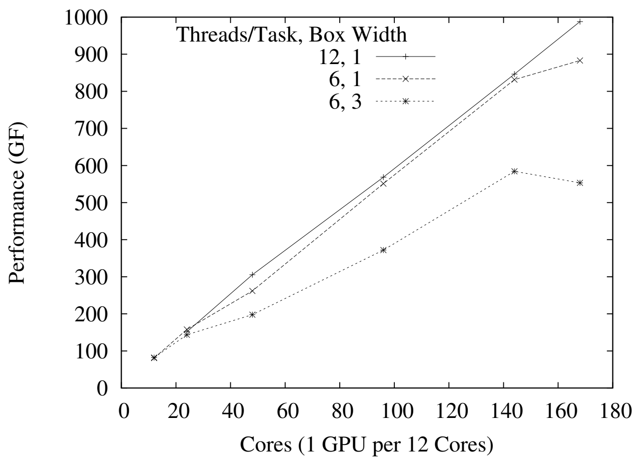

Figure 12. The performance of the CPU-GPU overlap implementation (Section IV-I) on Yona for various combinations of OpenMP threads per MPI task and box thickness.

Like for Lens, the best performance comes from few tasks per node, often just one task.

The best box thickness is often just one, so the CPUs are responsible for just a veneer of points around the GPU's domain. The reduction in box thickness on Yona versus Lens makes sense in general because the GPUs on Yona are a larger fraction of the total computational power than the GPUs on Lens. The remarkable thinness of the box, however, spread across all twelve CPU cores, indicates that load balancing is not the key feature of this implementation.

The key feature is instead most likely to be the decoupling of MPI communication and CPU-GPU communication that a veneer of CPU points provides. Notice from Figure 8 that the best GPU-resident performance on Yona is 86 GF. This implementation keeps all memory operations, including periodic-boundary exchanges, on the GPU. Using the CPUs for this boundary exchange—in other words, using the implementations in Sections IV-F and IV-G on one node—cuts the performance to 24 and 35 GF, respectively. The best CPU-GPU overlap performance on one node is 82 GF, with a box thickness of 3 with 2 tasks per node. The CPUs are not taking load away from the GPU as much as hiding the the cost of the CPU-GPU communication, and thus bringing performance back up to the level of the GPU-resident implementation.

VI. Conclusions

We presented performance results for various Fortran implementations of a three-dimensional linear-advection test case on four computers: a Cray XT5 (JaguarPF), a Cray XE6 (Hopper II), a multicore Infiniband cluster with NVIDIA Tesla C1060 GPUs (Lens), and a multicore Infiniband cluster with NVIDIA Tesla C2050 GPUs (Yona). We tested hybrid MPI and OpenMP implementations that perform bulk-synchronous computation and communication, that use nonblocking communication for overlap of computation and communication, and that use OpenMP threads for overlap. We found that attempting to overlap MPI communication with computation does not yield significant performance improvements for our test case, but tuning the number of OpenMP threads per MPI process does.

We also tested hybrid MPI, OpenMP, and CUDA implementations that attempt to overlap various combinations of CPU computation, GPU computation, MPI communication, and CPU-GPU communication. The implementation with the best performance partitions the task-local domain such that the CPUs compute a thin box around the block computed by a GPU. This implementation dramatically outperforms the other parallel implementations, by a factor of two or more. It is able to nearly match the per-GPU performance of a single-GPU implementation that keeps the problem in GPU memory. This performance comes at a cost in code complexity, however, with four times the lines of code of the single-process multithreaded CPU implementation.

For multiple nodes, the best performance of the CPU-GPU overlap implementation comes from a partition that gives minimal work to the CPUs. We conclude that the performance improvement does not come primarily from load balancing but from decoupling the MPI communication from the CPU-GPU communication.

Our results have implications for automatic tuning. We see a clear need to tune the number of threads per task. Our test has the additional tuning parameter of the thickness of the CPU box partition, which can itself depend on the number of threads per task. A potential dependence we did not test but which could be significant is the GPU thread-block size. The optimal size could vary with the size of the local domain on the GPU, which itself varies with the number of GPUs for strong-scaling cases like ours.

The GPU clusters we tested both have a substantial number of CPU cores per GPU. Because the CPUs perform minimal work in our best-performing implementation, a computer tuned for our test might have a smaller number of CPU cores per GPU, or conversely a larger number of GPUs. Targeting multiple GPUs per node is currently difficult using CUDA Fortran, but we do not expect this to be a long-term issue.

We note that a dominant factor in performance of current GPU clusters is the cost of CPU-GPU communication over a PCIe bus. An architecture with faster, lower-latency CPU-GPU communication could have a performance profile significantly different from what we see for Lens and Yona, with potentially significant implications for the design of CPU-GPU implementations.

Acknowledgment

We thank the anonymous reviewers for their helpful suggestions on improving our work. Portions of this work were funded by the Office of Biological and Environmental Research and the Office of Advanced Scientific Computing Research, both of the US Department of Energy. This research used resources of the OLCF at Oak Ridge National Laboratory and of NERSC at Lawrence Berkeley National Laboratory, both of which are supported by the Office of Science of the US Department of Energy.

References

[1] J. B. White III and S. W. Bova, "Where's the overlap? An analysis of popular MPI implementations," in *MPIDC99*, 1999.

[2] R. Brightwell, R. Riesen, and K. Underwood, "Analyzing the impact of overlap, offload, and independent progress for message-passing-interface applications," *International Journal of High-Performance-Computing Applications*, vol. 19, no. 2, pp. 103–117, Summer 2005.

[3] A. St Cyr and S. Thomas, "High-order finite-element methods for parallel atmospheric modeling," in *Computational Science—ICCS 2005*, ser. Lecture Notes in Computer Science, V. Sunderam, G. Van Albada, P. Sloot, and J. Dongarra, Eds., vol. 3514. Berlin: Springer-Verlag, 2005, pp. 256–262.

[4] A. Shet, P. Sadayappan, D. Bernholdt, J. Nieplocha, and V. Tipparaju, "A framework for characterizing overlap of communication and computation in parallel applications," *Cluster Computing*, vol. 11, no. 1, pp. 75–90, March 2008.

[5] R. Thakur and W. Gropp, "Test suite for evaluating performance of multithreaded MPI communication," *Parallel Computing*, vol. 35, no. 12, pp. 608–617, December 2009.

[6] P. Micikevicius, "3D finite-difference computation on GPUs using CUDA," in *2nd Workshop on General Purpose Processing on Graphics Processing Units*, 2009.

[7] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Conference on High-Performance Networking and Computing*, 2008.

[8] S. Kamil, C. Chan, L. Oliker, J. Shalf, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *IEEE International Symposium on Parallel and Distributed Processing*, 2010.

[9] R. Nath, S. Tomov, and J. Dongarra, "An improved MAGMA GEMM for Fermi GPUs," University of Tennessee Computer Science, Tech. Rep. UT-CS-10-655, July 2010.

[10] *CUDA Fortran Programming Guide*, The Portland Group, November 2009.

[11] P. Lax and B. Wendroff, "Systems of conservation laws," *Communications on Pure and Applied Mathematics*, vol. 13, pp. 217–237, 1960.

[12] *OpenMP Application Program Interface*, OpenMP Architecture Review Board, May 2008.