

Message Passing on Data-Parallel Architectures

Jeff A. Stuart

Department of Computer Science
University of California, Davis
stuart@cs.ucdavis.edu

John D. Owens

Department of Electrical and Computer Engineering
University of California, Davis
jowens@ece.ucdavis.edu

Abstract

This paper explores the challenges in implementing a message passing interface usable on systems with data-parallel processors. As a case study, we design and implement the “DCGN” API on NVIDIA GPUs that is similar to MPI and allows full access to the underlying architecture. We introduce the notion of data-parallel thread-groups as a way to map resources to MPI ranks. We use a method that also allows the data-parallel processors to run autonomously from user-written CPU code. In order to facilitate communication, we use a sleep-based polling system to store and retrieve messages. Unlike previous systems, our method provides both performance and flexibility. By running a test suite of applications with different communication requirements, we find that a tolerable amount of overhead is incurred, somewhere between one and five percent depending on the application, and indicate the locations where this overhead accumulates. We conclude that with innovations in chipsets and drivers, this overhead will be mitigated and provide similar performance to typical CPU-based MPI implementations while providing fully-dynamic communication.

1. Introduction

Coprocessors, particularly high-throughput, data-parallel coprocessors, can be incorporated with new, multi-core CPUs to create a powerful, heterogeneous computing solution. Special types of coprocessors, data-parallel machines (DPMs) such as Graphics-Processing Units (GPUs), are designed to give excellent performance with certain types of problems. Due to this specialization, DPMs tend to be used in one of two ways; single-DPM systems, such as a workstation with a single GPU, or multi-DPM systems. In either case, the only problems tackled tend to be those which can be statically divided into small chunks, and whose communication requirements are straight forward and can be statically determined. More detail is given in Section 2.

Our work lifts this problem-domain restriction. We modify the capabilities of coprocessors to allow dynamic communication between coprocessors and both CPUs and other coprocessors. Such communication is possible with very little loss of performance, and sometimes no loss.

To accomplish such a task, it is necessary to re analyze how communication happens in a data-parallel environment. Section 3 provides this analysis and discusses the design and implementation of a communication framework for GPUs. We call our framework DCGN, which is pronounced “decagon” and stands for “Distributed Computing on GPU Networks.” It is beneficial for the communication model to be both flexible and well-known. DCGN exports its capabilities via the send/recv and collective communications model from MPI. The interface to DCGN is not identical to MPI; DCGN has support for virtualizing communication targets across a process or DPM, something MPI does not. Some minor differences between DCGN and MPI are shown in snippets from a ping-pong application in Figure 3. DCGN uses an extension called “slots” to MPI. Slots allow a communication target (MPI rank) to be virtualized across multiple threads, both on a CPU and on a DPM. Figure 1 demonstrates a simple use of slots to virtualize a single GPU into multiple communication targets.

This new programming and communication model must not sacrifice performance for flexibility, or vice versa. To show that this trade-off isn’t necessary and that dynamic communication is indeed possible, we provide details on several benchmark applications in Section 4. The framework’s test results are presented in Section 5.

Section 6 presents a thorough discussion of the applications and observed results. Finally, we provide closing thoughts and a look towards the future and what avenues of work are newly opened in Section 7.

2. Background

Not only are GPUs and other DPMs powerful (the NVIDIA GTX 280 series of GPUs [4] are capable of sustaining nearly a TFLOP), they are relatively inexpensive and modular. High-performance GPUs use the PCI Express bus, available in desktop and rack-mounted systems. This modularity has helped to popularize clusters containing DPMs, such as those at Maryland, Terrasoft, and Los Alamos National Labs.

Clusters of CPUs and DPMs are just one example of heterogeneous computing. Upcoming single-chip heterogeneous processors (such as AMD’s Fusion and Intel’s Nehalem) may lead to future architectures with many types of

cores, and possibly more cores than can be powered on at the same time. This trend motivates further investigation into programming models that support heterogeneous processors. We submit that these heterogeneous cores must be first-class processors in communicating with other processors in the system, with autonomy from a primary CPU core, to achieve their full potential. DCGN represents a significant step towards this goal; it promotes GPUs to first-class computational resources.

2.1. The Graphics Processing Unit

The GPU is one of the most popular DPMs in use today. Modern GPUs are equipped with several multiprocessors, each capable of running their own unique code simultaneously, potentially achieving substantial performance gains. One significant limitation of the GPU is that its applications must conform to a stricter programming model than the CPU to achieve performance gains.

The first popular GPUs, designed for 3D graphics, used a pipeline that provided only fixed functionality. The recent addition of programmability to the GPU allows GPUs to target a broad range of application domains, not just 3D graphics. The modern GPU is no longer a 3D pipeline with some programmable elements; instead it is a programmable parallel processor with some 3D fixed-function hardware. NVIDIA's new GPU architectures is programmed via an extension to C called CUDA [3]. CUDA offers developers direct access to the GPU's programmable units and memory. The full scope of CUDA is beyond this paper, but NVIDIA publishes a very thorough programming guide [3].

CUDA allows data-parallel code to be run on GPUs, but only when told to do so by a CPU. When a CPU tells a GPU to run a kernel, that kernel is run using threads arranged in a multidimensional grid specified by the developer. A full grid is typically too large to run concurrently. Instead it is broken into a group of blocks, and each block is then separated further into an amount of threads capable of being ran concurrently on one of the multiprocessors. Blocks are not time-sliced; once a block is scheduled onto a multiprocessor it must finish execution before another block may take its place.

CUDA kernels are not capable of managing GPU memory; this must be handled by the CPU. The GPU also cannot communicate with any devices on the PCI-e bus, including network cards. Thus, the execution of a CUDA kernel without DCGN typically follows a model of: allocate GPU memory via the CPU, copy CPU memory to GPU memory via the CPU, modify GPU memory with one or more kernels, and finally copy results from GPU memory to CPU memory via the CPU. This push/pull paradigm, while well-suited for many application scenarios, does not allow for communication between kernels and the CPU while a kernel is running.

2.2. Message Passing Interfaces

Massive multiprocessor machines allow for easy inter-process communication (IPC). Primitives such as shared memory segments and semaphores can be used for fast and efficient synchronization and communication. As clustered computing became more commonplace, a standard for IPC over networks was developed. The Message Passing Interface (MPI) [11] came to fruition in the mid-1990s.

The stated goals of MPI are high performance, scalability, and portability. Achieving low latency and high throughput is important for computing clusters, since a lack of shared memory implies large amounts of network data transfer. Portability is very important for MPI. Scientists often purchase new clusters as grant money allows, and clusters can differ in architecture or network interface from one generation to the next. Scientists have no desire to rewrite large amounts of code every time a new cluster with a different architecture is purchased. The scalability of MPI is primarily due to MPI being the de facto standard in distributed computing, as well as the lack of any architecture-specific functionality in MPI. Supercomputer and networking manufacturers understand that sales depend on an available MPI implementation.

2.3. Distributed Computation using GPUs: State of the Art

Most of today's GPU research involves one CPU and one GPU; distributed computation is a difficult challenge. The ideal method would yield both performance and flexibility, supporting performance speedups across many nodes as well as fully dynamic communication and a choice of programming models. Currently, such a method does not exist.

Two vendor-supplied GPU communication schemes are NVIDIA's Scalable-Link Interface (SLI) [2] and ATI's Crossfire [1]. Neither of these methods allow for general-purpose or programmable communication; they simply make multiple GPUs appear as one to the application.

The research community has tended to lean towards one side or the other. Two recent research projects, Zippy [5] and CUDASA [12], use the Global Arrays [9] paradigm to achieve excellent performance. However, both require static communication patterns and thus force developers into one specific programming and communication model. Merschell and Owens tried the opposite approach [8] with an OpenGL-based implementation of a multi-GPU distributed-shared memory (DSM) system to allow for maximum flexibility. However, due to limitations of the API, current GPUs, and the nature of DSM, their implementation suffered severe performance problems.

Developing a general-purpose, high-performance inter-GPU communication scheme presents many difficult chal-

lenges. Perhaps the major hurdle is the traditional use of the GPU as a coprocessor under the direction of a CPU host. This model of GPU-as-slave (GAS) restricts the programmer in many ways. Each GPU only communicates with its host CPU. Implementations of this model tend toward static communication and partitioning of the input space or problem space, often times implementing Global Arrays. GAS approaches are well-matched to many computational patterns but are limited by the lack of dynamic communication and the dependence on a host CPU.

One typical GAS method is to statically divide work into N parts, send each part to a free GPU, receive work when a GPU finishes, and repeat until all work completes. This is an effective and efficient method for static datasets. For any problem that does not have a statically-known size, or for any problem with data dependencies, this method is very hard if not impossible to employ in an efficient manner.

Another GAS method involves dividing the task domain into N parts and then connecting those N parts into a pipeline. Data is given to the first set of GPUs, which then all perform the same stage of a pipeline. When the first set finishes a piece of data, the data is shipped to the second set of GPUs for processing and more data is given to the first set of GPUs. This method works well on certain types of data, but as with the method discussed in the previous paragraph, this method does not extend well to problems poorly suited to pipelining.

3. Communication on Data-Parallel Architectures

Message passing on data-parallel architectures and heterogeneous computing platforms has a unique problem set and philosophy; principles from CPU-based message passing do not necessarily transfer over. Perhaps the largest problem for GPU-based message passing is to maintain flexibility *and* dynamic communication. Developers shouldn't be forced into a single model of communication and/or computation. While CPUs serve as first-class computational and communication resources simultaneously, GPUs do not. This impedes the progress of multi-GPU development; our design addresses this inability.

3.1. Challenges with MPI and data-parallel coprocessors

MPI, the de facto standard API for messaging in supercomputing, is a peer-to-peer API that provides efficient and highly optimized communication methods while not relying on a specific network interconnect. Networking vendors develop new interconnects, write an MPI implementation, and allow scientists to compile and run previous MPI codes with the new networking hardware.

MPI implementations assume that each of a job's processes are communication targets with direct access to communication hardware. This is not the case on DPMs; they yield a set of non-traditional challenges when implementing an MPI. DPMs can't source communication as they often don't have access to communication hardware. CPUs must initiate communications with DPMs by either pushing to or pulling from DPM memory. Two methods (each with their own limitations) are used to communicate with a DPM: access DPM memory inbetween kernel invocations, or poll DPM memory while a kernel is running. The former disallows dynamic communication, the latter suffers from high CPU load and wasted cycles.

To make a useful DPM MPI, we first looked at how CPUs and DPMs differ. DPM threads have large amounts of parallelism with many short-lived, lightweight threads per kernel. Threads are grouped, often by function and/or memory coherence. CPU threads are autonomous from other threads and have their own stack, registers, and instruction pointer. CPU threads perform MIMD computations, DPM threads often perform SIMD computations. Even though CPU threads are autonomous, MPI treats processes, not threads, as individual communication targets; each process gets a rank, not each thread. DPMs have no concept of a process, and thus there is no clear mapping of ranks for DPMs. The typical mapping of one rank per process (for a DPM, this would be a kernel) doesn't always fit best. Nor does one rank per thread because of complications that arise with collectives.

Instead of forcing a DPM MPI into one mapping, we chose a robust approach that uses a paradigm we call *slots*. Ranks are virtualized N-ways across a DPM based on the number of slots requested for a specific DPM by the user at the start of the job. Each DPM has at least one slot. The maximum number of slots is equal to the maximum number of threads that are simultaneously executed (again, this is tied to limitations introduced by collectives). As DPM algorithms can vary wildly, slots give the developer the flexibility to find the best mapping for their algorithm.

Slots breakaway from the MPI standard. Because of this, we chose not to extend MPI, and instead to create our own, similar API. As MPI is a committee based standard, we felt it best to let the committee take what they will from this research and we hope that the notion of slots will find it's way into MPI. For those concerned with the extra overhead in porting old codes, those codes would have to be completely rewritten for DPMs, and the added task of a few find-and-replaces was minimal by comparison.

To motivate slots, consider two applications. The first is a parallel implementation of map-reduce. Billions of elements need to be reduced. Every element requires exactly X nanoseconds to process, and the work-master ensures that each DPM receives as many elements per request as there exist available threads on the DPM. It makes complete sense

```

const int SLOT_INDEX = 0;
if (dcgn::gpu::getRank(SLOT_INDEX) == 0) {
  if (threadIdx.x == 0) {
    dcgn::CommStatus stat;
    // note that for communication, we have to use global
    // memory. this is a byproduct of the memory system
    // on the GPU.
    dcgn::gpu::send(SLOT_INDEX, 1, gpuMem, gpuMemSize);
    dcgn::gpu::rcv(SLOT_INDEX, 1, gpuMem, gpuMemSize, &stat);
  }
} else if (dcgn::gpu::getRank(SLOT_INDEX) == 1) {
  if (threadIdx.x == 0) {
    dcgn::CommStatus stat;
    dcgn::gpu::rcv(SLOT_INDEX, 0, gpuMem, gpuMemSize, &stat);
    dcgn::gpu::send(SLOT_INDEX, 0, gpuMem, gpuMemSize);
  }
}
__syncthreads(); // barrier for all threads in block.

```

Figure 1: A snippet from a GPU ping-pong application written using DCGN. A ping-pong application sends data from host A to host B, followed by host B sending data back to host A.

to have one slot per DPM in this example as communication costs are reduced. Now change this example slightly. Assume that 0.001% of the elements require 10,000X nanoseconds to process. A single element can then delay an entire DPM from communicating results, as virtually every thread is left idle while the time-intensive element is being processed. It makes sense to allocate extra slots to DPMs, perhaps based on the number of multiprocessors present. These examples demonstrate that no single mapping of ranks to DPM resources can match every data parallel algorithm’s requirements and that an approach must provide a flexible level of granularity.

3.2. Case Study: Communication across GPUs

Using NVIDIA G92 GPUs and the notion of slots, we implemented a library capable of allowing GPUs across a cluster to communicate with each other and with CPUs on the same cluster. The library, Distributed Computing on GPU Networks (DCGN), offers fully dynamic communication capabilities, just like MPI. Just like with MPI, any communication target may communicate with any other communication target, either through point-to-point communication or through collectives. To illustrate the flow of a send from a GPU on one node to a GPU on another node, we present an example in Figure 2.

Dynamic communication is a very important component of DCGN, as is granting full access to CPU and GPU resources. DCGN employs kernels as its computing primitive, these may be run either on the CPU or the GPU and have complete access to both. Developers are responsible for kernels, and they write for the GPU and CPU as desired. DCGN will not automatically convert a CPU kernel to a GPU kernel. Kernels are launched via calls to DCGN. As kernels issue communication requests, DCGN relays the requests and performs the communication in a manner hidden from the user. We felt that allowing full use of compute resources and fully dynamic communication was an important approach; instead of forcing developers to

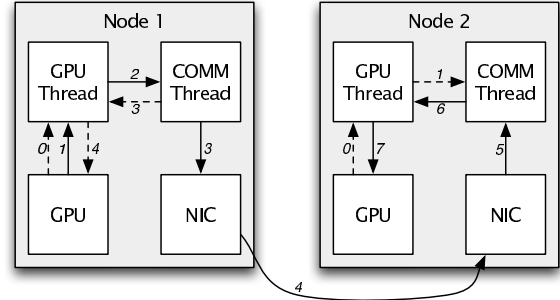


Figure 2: Dataflow of a send from one GPU to a GPU on another node. The numbers accompanying each arrow dictate the order in which events happen. The solid lines indicate paths where actual send-data is transferred. The dashed lines indicate paths where control messages (e.g., handshaking, polling data) are passed. Identical time indications do not necessarily mean concurrent events. Time indications serve to order events local to their path. The send commences as Node 1 polls the memory of the GPU (0) and finds a send-request. Meanwhile Node 2 polls the memory of the GPU and finds a receive-request. Next, Node 1 reads the requested send-data from GPU memory (1) while on Node 2 the receive-request is packaged and relayed to the COMM thread. Node 1 then finishes reading the data for the send-request, packages it, and sends it to the COMM thread (2). The COMM thread executes and finishes the MPI call by transferring data to the network interface (NIC) (3) and signaling the CPU to inform it that the send completed. The send-data is transmitted from the NIC on Node 1 to the NIC on Node 2 (4), as well as the CPU on Node 1 signaling the GPU that the send completed locally. Next, the send-data is received by the COMM thread on Node 2 (5). The send-data is then copied to the GPU thread (6). Finally, the send-data is copied to the GPU (7) and the GPU is signaled to let it know the receive completed.

use the GPU with a fixed programming and communication model (as in Zippy [5]), we view DCGN as a substrate upon which developers construct a variety of programming and communication models.

The communications interface of DCGN is similar to MPI as both allow fully dynamic communication and share similar features. However, MPI is large and complicated and has many features not present in DCGN. GPUs are one type of computational target of DCGN. A “slot-identifier” parameter is present in most GPU-API functions of DCGN, but not in MPI. Kernels pass this slot-identifier to enforce explicit mappings of GPU-sourced communication requests to slots. DCGN’s communication model is still very similar to that of MPI, as can be seen in the comparison of a DCGN code snippet and its corresponding MPI code, shown in Figure 3.

DCGN not only allows kernels to run on the GPU and

```

int x;
MPI_Status stat;
if (rank == 0) { // send and recv with rank1.
    MPI_Send(&x, 1, MPI_INT, 1, 0,
             MPI_COMM_WORLD); // send ping
    MPI_Recv(&x, 1, MPI_INT, 1, 0,
             MPI_COMM_WORLD, &stat); // recv pong
} else if (rank == 1) { // recv and send with rank0.
    MPI_Recv(&x, 1, MPI_INT, 0, 0,
             MPI_COMM_WORLD, &stat); // recv ping
    MPI_Send(&x, 1, MPI_INT, 0, 0,
             MPI_COMM_WORLD); // send pong
}

-----

int x;
dcgn::CommStatus stat;
if (dcgn::getRank() == 0) {
    dcgn::send(1, &x, sizeof(int));
    dcgn::recv(1, &x, sizeof(int), &stat);
} else if (dcgn::getRank() == 1) {
    dcgn::recv(0, &x, sizeof(int), &stat);
    dcgn::send(0, &x, sizeof(int));
}

```

Figure 3: A snippet from a ping-pong application written using MPI (top) and DCGN (bottom). A ping-pong application sends data from host A to host B, followed by host B sending data back to host A.

kernels to communicate in a fully dynamic fashion, it also eliminates the “middle-man (CPU)” in application code. DCGN implicitly handles all communication requests in such a way as to shield the developer from the underlying details. DCGN even goes a step further in allowing the possibility for only GPU application code to be run. If a developer so chooses, and we believe developers often will, no CPU kernels need be run. All kernel code may run on the GPU. This is an important step, as kernels will see a large speedup if and when GPUs bypass the CPU and directly communicate with the network interface card (NIC).

3.2.1. Design Goals of a DPM-Communication Library.

The two high-level design goals of a DPM-communication library should be *performance* and *flexibility*. Lower level goals also exist and aid in the implementation of the two high-level goals.

Modern PCs and cluster nodes are equipped with multi-core CPUs. MPI only treats processes, not threads, as first-class communication sources. To take full advantage of a multi-core and/or multi-CPU machine, MPI must run one process per core, or the developer must be content to have all threads share the same rank. *It is essential for a modern communication library to better utilize all the cores on a CPU without resorting to heavy weight processes and inter-process communication.*

Modern computers contain multicore CPUs and/or multiple GPUs, they can also contain multiple GPUs. However modern graphics drivers use a one-to-one mapping of CPU threads to GPUs. *A library used to control both computation and communication should be able to efficiently handle all GPUs, as well as make launching computation kernels on all GPUs a simple task.*

GPU drivers are not the only pieces of software that potentially are unsafe for multiple threads. Many implemen-

tations of MPI either do not guarantee thread safety, or they guarantee thread safety with a performance penalty. A high-level communication library for DPMs is likely to leverage an existing MPI implementation. *It is then very important that safe, concurrent access be guaranteed by the higher-level library, as such access is not always guaranteed by MPI implementations.*

Along with guaranteeing safe access to the MPI implementation, *it is also important to fully leverage the power of the underlying MPI implementation.* Most implementations contain highly-optimized versions of collective communication routines (e.g. broadcast, scatter). One could easily write their own version of these collectives, but shrugging off the research and fine tuning already done by others would be asinine.

Many DPMs are not capable of pushing data to a CPU. This makes sourcing communication a challenge. This challenge must be overcome by any worthwhile communication library though, as having no ability to source communication from all computational entities severely restricts developers.

3.2.2. DCGN Architecture. With these design goals in mind, we created DCGN, a multithreaded communication library. DCGN takes advantage of the multithreading capabilities of modern CPUs by spawning many internal threads. Thread-safe queues are used to control inter-thread and inter-node communication.

Internal DCGN threads are assigned specific tasks, and stay alive for the life of the application. Each thread owns a work queue, into which requests such as kernel launches or communication calls are funneled. DCGN uses MPI as its underlying communication library and executes computational kernels on the CPU and GPU. DCGN allows for fully dynamic communication between any combination of CPUs and GPUs.

DCGN supports three types of threads, each based on the responsibilities it holds. CPU-controlling threads execute kernels on the CPU and funnel communication requests from CPU kernels to the communication thread. In this manner, DCGN takes care of the details behind launching computational threads.

DCGN threads that control a GPU execute kernels on the GPU, monitor the GPU for communication requests, transfer memory between the CPU and GPU, and funnel communication requests from GPU kernels to the communication thread. As DCGN handles the GPUs, the developer need to perform any explicit GPU management. Developers request a number of GPUs for a specific machine and let DCGN handle the details. They also do not need to split kernels across communication calls as DCGN allows GPUs to source communication via normal function calls like *send* and *recv*. Certain facilities such as file I/O are not present in CUDA (and thus not yet in DCGN). While this prevents full autonomy from CPUs, all the demanding parts of scientific

computations can be handled by user-developed DCGN-kernels for the GPU, thus giving the GPU near-complete autonomy from the CPU.

Another benefit of this design is that applications that perform both CPU and GPU computations no longer need to be tied together in one big piece of code. Kernels and GPU kernels may be separated; DCGN handles all requests to launch kernels and requests for communication from kernels on the GPU.

The last class of threads are communication threads. The communication thread initializes the underlying MPI, handles communication requests from kernels, signals CPU- and GPU-controlling threads as communications complete, and shuts down the underlying communication library upon application completion. Each DCGN process spawns exactly one communication thread. This method allows DCGN to provide thread-safe access to any communication library, even a potentially non-threadsafe implementation of MPI. Figure 4 shows the typical DCGN instantiation on a single node.

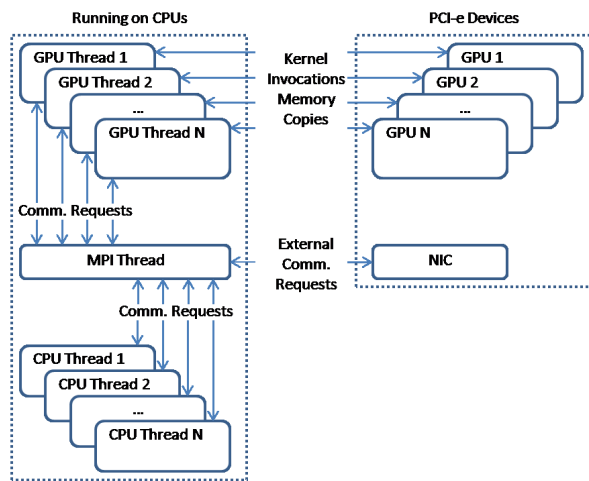


Figure 4: The typical architecture of a DCGN process running on a single node. The CPU thread(s), GPU thread(s), and MPI thread all run on the node’s CPU(s). The GPU(s) and NIC are physically connected via the PCI-e bus. Each GPU threads issues kernel requests and memory copies to its respective GPU. Each CPU thread and GPU thread relay communication requests to and from the MPI thread. The MPI thread communicates data to and from other nodes via the NIC.

To illustrate a real-world instance of the DCGN architecture, consider a small cluster with four nodes. Each node has two CPUs, each with two cores, and each node also has two GPUs. A developer decides to use a homogeneous setup with all nodes using two CPU-kernel threads per node and two GPU-kernel threads per node. A communication thread is implicitly created on each node as well. Therefore, each node has five threads: two threads control CPU kernels,

two threads control GPU kernels and monitor GPUs for communication, and one thread controls all inter-node and intra-node communication. A total of twenty threads are running on the four nodes.

3.2.3. DCGN Implementation. The implementation of DCGN posed several original challenges. First and foremost, the GPU must become a network-capable device. Secondly, the GPU must become (pseudo-) autonomous from the controlling CPU. These two challenges, coupled with the goals from Section 3.2.1 (utilize all cores, efficiently handle GPUs, guarantee safe concurrent access, and fully utilize underlying libraries) shaped the final implementation of DCGN.

DCGN takes advantage of multiple cores by multiplexing MPI ranks across all requested CPU cores and light-weight GPU threads; a DCGN node with a certain configuration could easily multiplex a single MPI rank one-thousand ways. As DCGN multiplexes these ranks, there needs to be a way for the DCGN library to know the virtualized rank. MPI uses global variables to achieve this task, DCGN uses thread-specific data (TSD). Internal DCGN data structures are allocated on the heap, and then stored for lookup. TSD allows for a fast, flexible, and cross-platform method of referencing DCGN information.

Part of the TSD stored for DCGN is whether a specific thread controls outside communication via MPI (comm. thread), runs CPU kernels (CPU-kernel threads), or controls a GPU (GPU-kernel threads). Not every MPI implementation is thread- safe; DCGN avoids potential problems by having a single thread interact with MPI; this thread initializes and finalizes MPI, and performs all communication calls (e.g. *MPI_Recv*). CPU-kernel threads and GPU-kernel threads use a thread-safe queue to relay communication requests to the comm. thread. CPU-kernel threads are simply in charge of invoking CPU-based kernels and relaying communications. GPU-kernel threads are responsible for initializing and finalizing a GPU, invoking kernels on the GPU, and monitoring the GPU for communication requests and handling them appropriately.

The GPU-kernel thread was the largest challenge because GPUs can’t source communication. Kernels are currently split across points of communication. DCGN alleviates this problem by allowing direct communication calls (e.g. *send*, *recv*) in GPU kernels. These calls don’t interact with the network driver; they set regions of GPU memory that are monitored by a GPU-kernel thread. When the memory is noticed, the request is obtained via *cudaMemcpyAsync*, handled, and the appropriate memory is set on the GPU to flag the GPU kernel, telling it to continue execution.

Side effects arise from CPUs controlling GPUs; messages are not instantaneously transferred by GPUs, the CPU must implement a polling scheme, and a trade-off between low-latency messages and efficient CPU-utilization must be

made. Messages have to be polled from a GPU; this requires several rounds of PCI-e transfers. The CPU must poll the GPU at a certain interval since the GPU can't signal the CPU. Tradeoffs in performance are required because high-frequency polling strains the CPU whereas low-frequency polling increases message latency.

It is desirable to run MPI codes as many MPI algorithms already exist. This raises two issues; mapping MPI calls to DCGN calls, and automatically assigning the number of slots to a GPU. The first issue is straightforward. The second issue is non-trivial and beyond the scope of this paper. The simplest static mapping of slots would be a one-to-one slot-to-GPU pairing. This is not always optimal and it discards the entire notion of slots.

While the automatic allocation of slots is a hard problem, assigning virtualized ranks to CPU-kernel threads and slots is easy. Every $Node_n$ is given $C_n + (G_n \times S_n)$ ranks, where C_n is the number of CPU-kernel threads requested, G_n is the number of GPUs requested, and S_n is the number of slots per GPU requested. Ranks are assigned consecutively within a node, and in increasing order across successive MPI ranks. The lowest non-issued rank is given to the first CPU, then the second, and so on. Then slot 0 on GPU 0, then slot 1 on GPU 0, and so on, until all CPUs and GPU slots are assigned virtualized ranks. There's no implicit assumption of ranks per node, and threads from different processes on the same node have no explicit interaction.

Use MPI collective routines with multiplexed MPI ranks

All DCGN collectives follow a similar pattern and differentiate themselves upon calls to MPI. As local requests for collectives trickle in from CPU kernels and GPU kernels, these requests are stored. Once all CPU kernels and GPU kernel slots initiate the collective, data is massaged as needed and then the MPI function is called. Upon completion of the MPI function, data is copied as necessary to local buffers, then each CPU kernel and GPU kernel slot is notified of the communication completion.

From an individual node's perspective, when a broadcast is called, the root of the broadcast is either resident on the node or on a different node. In the case of the root being resident on the node, the MPI broadcast is executed using the root's specified buffer. If the root is not resident on the node, one buffer is selected at random from those specified for use with the MPI broadcast. In either case, upon completion, the memory is copied from the buffer given to MPI to all the other buffers. One optimization intended for the future is to have memory copies happen in a tree-like manner across CPU and GPU threads. This would drastically cut the local time taken for large broadcasts. It would also significantly alleviate bandwidth to the GPU if multiple slots per GPU were used.

Other collectives come with more challenges. DCGN does not implement all collectives found in MPI. However

we propose a general pattern for use with gather, scatter, and all-to-all. Homogeneous node configurations simply require an invocation of the MPI collective with all the representative data for the entire node, followed by a dispersal to the appropriate CPU threads and GPU slots. Heterogeneous configurations work in much the same way, but as data sizes can differ from node to node, the vector variants (e.g. MPI_Scatterv) should be used.

3.2.4. DCGN Limitations. As of right now, developers using DCGN must deal with a few limitations. Our design anticipates mitigating these limitations with future improvements to GPU and chipset drivers. Currently, CPU kernels cannot directly use MPI but instead must use DCGN primitives to communicate. User-spawned CPU-threads cannot call DCGN communication functions. Certain GPU kernels have limitations as to the number of blocks that can be scheduled. Communication currently is not as efficient when sourced from the GPU as it is when sourced from the CPU.

Developers are not allowed to directly call MPI functions. At this time, MPI is used under the hood, but that may not always be the case. Also, DCGN relies on MPI being in a stable and expected state. This may not be the case if a developer manually changes the state of the MPI implementation. This restriction may be lifted as advances in MPI and/or GPU drivers are made.

Users have the ability to spawn their own threads on the CPU beyond the number of threads supplied by DCGN. Communication from these threads via DCGN is not allowed. This is because DCGN does not have any knowledge of these threads, and thus no thread-specific data for communication is stored.

GPU kernels that use DCGN communication are limited in the number of blocks that can be scheduled onto the GPU; The number capable of being scheduled onto the GPU is limited by the hardware, and once a block is scheduled to a multiprocessor, it runs until completion. The order of block scheduling is arbitrary, and thus if one expects a single block (e.g. block 0) to perform communication before all other blocks can perform computation, a deadlock will occur if all multiprocessors are taken before that block can be scheduled.

Communication involving a GPU is not as efficient as communication only involving CPUs. Two primary reasons cause this inefficiency: GPU memory must be polled to check for communication requests and the CPU must act as a relay between GPU memory and the NIC. Small data transfers are impacted more than large data transfers as initialization time is much larger compared to transfer time with respect to a small amount of data.

These limitations may seem constricting, but empirical evidence shows this is not the case. DCGN strives to provide communication functionality so as to mitigate the need to call MPI functions. If a developer want more

communicatoin- capable CPU threads, they simply request more CPU-kernel threads. GPU kernels can be wrapped in a loop to get around the limitation with respect to number of blocks. Finally, GPU developers may not achieve ideal speedups because of communication overhead, but we believe vendors will add functionality to chipsets, GPUs, and drivers to eliminate the need for the CPU to mediate between the GPU and NIC.

3.2.5. DCGN vs. PGAS. With a PGAS implementation, code is ran with a hierarchical address space. Algorithms rely on implicit copies of memory from a neighboring node to local memory. Communication costs are determined statically at compile time because the communication is simple. This predictability is a strength as it allows for optimizations such as pre-fetching. However it limits users in that efficient implementations of more complex communication patterns, especially collectives, are not possible.

With DCGN, just as with MPI, communication can be dynamic, and highly optimized routines can be used. Users must extend their own energy to do things such as pre-fetching and overlapping communication with computation. However this extra effort at the trade-off of optimized communication is common in modern distributed applications.

4. Test Applications

To fully test DCGN, we wrote several applications. These exercised throughput and latency for point-to-point and collective communications. We tested GPU GPU, CPU GPU, GPU CPU, and CPU CPU communication, as well as several communication models including fully dynamic and simultaneous communication.

It's an easy task to pick applications that are easily parallelizable, applications that have minuscule amounts of communication and lots of computation. Such applications can have perfect speedup. We avoided these applications and instead chose challenging applications that would thoroughly test DCGN's abilities and give a good measure of real-world performance. These applications either have non-trivial communication patterns or communication requirements that can't be overlapped with computation and have a significant effect on parallel efficiency.

The test applications were run on a cluster of four nodes, each with two dual core AMD Opteron(tm) 2216 processors with 1 MB of cache, 4 GB of RAM, and two G92 NVIDIA GPUs with 512 MB of GPU memory connected via PCI e. The machines run 64 bit Gentoo Linux 2.6.24 r3 with SMP support. The underlying MPI implementation was MVAPICH2-1.0. The nodes were connected via infiniband. This configuration allowed for a total of 8 CPUs and 8 GPUs meaning 16 total computation units. This number is less than ideal, but access to a machine with at least 32 G92 GPUs was not possible.

Sending and Receiving Simple send-and-recv applications were implemented to test throughput and latency of point-to-point communications. We implemented tests for both small (including zero-sized) and large message sizes. We also implemented tests that used multiple slots per GPU to understand the behavior of our system with respect to latency.

One-to-All Two applications were implemented to analyze one-to-all (broadcast) performance. The first test executed broadcasts of varying sizes. The second application implemented was a brute force N-body simulation [10]. N-body applications simulate the movement of celestial bodies based on the laws of gravitational physics. Given N bodies and P processors, the distributed algorithm works by each processor accumulating the force of all N bodies on $\frac{N}{P}$ bodies. As all bodies affect the movement of other bodies, a brute force algorithm requires $O(\frac{N^2}{P})$ time per time step for computation. Once all forces are calculated and applied, each communication target broadcasts its updated bodies to the rest of the targets. This process is repeated for an arbitrary number of time steps.

Simultaneous Communication Cannon's matrix multiplication [6] is a distributed algorithm that performs dense matrix multiplication of two $N \times N$ matrices, A and B , and computes the matrix $C = A \times B$. The algorithm uses P communication targets and runs in $O(N^2 \sqrt{P})$ time. P must be a perfect square, and $N \bmod \sqrt{P}$ must be 0. Cannon's algorithm works by arranging the targets in a square grid and distributing chunks of each matrix to the targets. Each target computes a chunk of C by performing incremental multiplications of chunks from A and B . After each sub-multiplication, the chunks of A and B are rotated among the processors. This is done using a call similar to `MPI_Sendrecv_replace`, and happens for each communication target nearly simultaneously. This algorithm has known communication costs and can thoroughly stress any network given large enough matrices.

Unpredictable Communication Calculating the Mandelbrot set [7] is an excellent candidate for testing dynamic and unpredictable communication. An iterative, per-pixel algorithm is used to generate an image of the Mandelbrot fractal. The number of iterations varies upon the location of the pixel in the problem domain. There is no shared data in the calculations of pixels. While global communication costs can be calculated, per-target communication costs cannot be calculated with one-hundred-percent accuracy. We implemented a dynamic work queue with GPUs to generate a Mandelbrot fractal. As GPU processors become available they contact the master thread (target 0) and request a strip of the output image to generate. The master responds by allocating a strip and marking that strip as *in progress*. Once a target is done with a strip, it returns the calculated strip to the master. Network and device latency yield unpredictable

and truly dynamic communication. Figure 5 shows how running the application twice with the same parameters can produce a different work distribution.

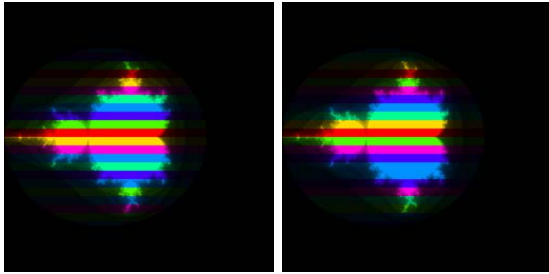


Figure 5: *Two separate runs of a Mandelbrot fractal generator using eight DCGN ranks. Horizontal strips are color-coded to show which ranks computed which sections.*

5. Results

DCGN proved to be a formidable asset when developing GPU and hybrid CPU-GPU applications. DCGN layered a fast, efficient, and familiar abstraction on top of the low-level communication primitives of the GPU. The ability to focus on the high-level algorithms instead of the low-level communication primitives made for an anecdotally faster and more productive development cycle.

5.1. Results from High-Level Applications

Not only does DCGN provide an efficient layer for writing applications, it is capable of yielding performant applications on par with those written using the GAS+MPI model. All three test applications obtained at least 90% of the performance of the corresponding single-node GAS implementations (the CPU didn’t do any work).

Why should DCGN be used at all if its performance is consistently below GAS+MPI approaches? The answer is two things: application codes that use DCGN will be future-proof as advances in GPU, chipset, and driver technology will be integrated into DCGN and the exact same application code will suddenly run faster. The other reason is that with DCGN, one can write code at a higher level and focus on the algorithms, not the communication requirements. As an analogy, developers can write their applications using BSD sockets or the Infiniband API, this may very well run faster than the corresponding MPI code. However, very few would ever do this as it would imply more effort, more lines of (potentially buggy) code, and less portability.

Mandelbrot The Mandelbrot test application, at peak performance, computed more than seventeen million pixels per second with a GAS implementation and more than fifteen million pixels per second with a DCGN implementation. With eight GPUs, the GAS implementation yielded

a speedup of $3.08\times$ and a parallel efficiency of 38% (where efficiency is defined as the speedup obtained with N computational units divided by N), while the DCGN implementation yielded a speedup of $2.72\times$ and an efficiency of 34%. Anecdotally, our DCGN implementation contained 25% fewer lines than our GAS+MPI version.

These numbers show how communication costs can dampen the results of any GPU application. Instead of allowing a single GPU to run the algorithm over the entire image, partial results are messaged back to a master. As dark pixels (those that converged quickly) took virtually no time to compute, large portions of the run time were dominated by communication. The DCGN version performed almost as well as the GAS version, with the discrepancy due to DCGN’s higher overhead in communication.

Matrix Multiplication The matrix-multiplication algorithm of Cannon makes use of the send/recv function in both DCGN and MPI. With a matrix size of 1024×1024 and four GPUs, the DCGN implementation yielded an efficiency of 71%, while the GAS implementation yielded an efficiency of 74%. Implementing a send/recv function in DCGN, instead of forcing users to manually use asynchronous sends and receives, helped performance. Not having to perform two GPU-memory polls and have two outstanding communication requests allowed DCGN to more efficiently handle communication, thus bringing the performance almost to the level of the GAS+MPI implementation.

N-body The N-body simulation allowed us to thoroughly examine the efficiency of collectives with both small and large message sizes. Both the DCGN and GAS implementations yielded the same efficiency when compared with the single-GPU implementation. This showed us that when computation severely outweighs communication, DCGN is just as powerful as GAS+MPI. Using eight GPUs and four thousand bodies, we obtained an efficiency of 28%. But when the number of bodies was increased to sixteen thousand, the efficiency rose to 64%. Efficiency peaked at just over 90% with thirty-two thousand bodies.

5.2. Results from Low-Level Micro Benchmarks

GPUs yield great speedup for many computations, but in a multi-node environment, they have fewer features and more overhead when communicating with other processors in the system. GPUs have many multiprocessors, so it is no surprise that decomposable tasks would execute significantly faster. But since GPUs are only connected to the CPU through a bus, and the two do not share memory, communicating data from a GPU on one node to a CPU or GPU on another node incurs more overhead. This is regardless of whether one uses the GAS or DCGN models.

DCGN works to alleviate this overhead in many ways. DCGN grants access to GPUs and provides abstract com-

munication such that advances in certain technologies are integrated with no effort on the developer. They see performance gains for free. This is important because DCGN currently yields good performance with both point-to-point and collective communications, but the performance could stand to improve. In particular, the higher overheads of DCGN compared to MPI+GAS leads to a slight performance degradation, as we described in Section 5.1. We now examine these overheads in more detail.

Point-to-Point Communication Point-to-point communication with DCGN was measured using micro-benchmarks of sends, receives, and ping-pong tests. We performed several iterations of communication with varying sizes of data to gather comparable numbers for communication speeds. For example, to compare broadcast speeds, timings were taken on the root node with a series of iterations per data size, with data sizes ranging from one byte to sixty-four megabytes. The time taken by `MPI_Bcast`, `dcgn::broadcast`, or `dcgn::gpu::broadcast` was recorded and compared across MPI, DCGN CPU, DCGN GPU, and DCGN CPU + GPU. The results for all four were similar, we discuss sends here. Small messages give poor performance with DCGN when compared to MVAPICH2. A zero-byte, CPU-CPU message with DCGN takes more than $28\times$ as long to deliver than with MVAPICH2, and GPU-GPU message took $564\times$ longer than with MVAPICH2. These large differences are due to the nature of DCGN and the GPU. DCGN uses a multi-threaded architecture with thread-safe work queues. When compared to the time it takes to simply call `MPI_Send`, it becomes clear why a CPU-CPU send takes longer. The GPU-GPU send can also be explained in terms of DCGN overhead. Three separate communications with the source GPU must take place. The CPU polls GPU memory, the CPU copies the appropriate memory from the GPU, and once the message is handled (which can take time), the CPU tells the GPU that the message was sent. All three stages require a significant amount of time.

As message sizes increase, the performance of DCGN nearly matches that of MVAPICH2. A 1 MB CPU-to-CPU send with DCGN takes only 4% longer than with MVAPICH2. To send a 1 MB GPU-to-GPU message with DCGN takes $1.5\times$ longer than a CPU-to-CPU message with MVAPICH2. The reason for the drop in discrepancy is that actual message transfer time far outweighs the initialization time for each stage of communication. A complete graph of the `send` micro benchmark is shown in Figure 6.

Collective Communication Even though optimizes DCGN for collectives, MVAPICH2 collectives tend to be faster. Collectives are built upon the send and receive primitives so collectives of a certain size show some of the same tendencies as point-to-point communication.

Barriers in DCGN took much longer to complete than with MVAPICH2. This is primarily because almost no data

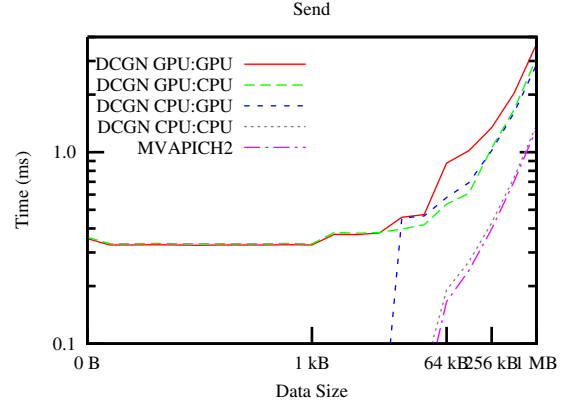


Figure 6: Sends for CPUs and GPUs with and without DCGN. The graph shows the time taken to send messages of various sizes. MVAPICH2 provides a baseline of performance. CPU-only DCGN timings perform very close to MVAPICH2 applications for large messages, where as small-message timings are dominated by the overhead of thread safety. Communication involving the GPU adds extra time as memory copies must be made between the CPU and the GPU.

is sent with a barrier. Almost all communication time (both in DCGN and MVAPICH2) is initialization. A full table of timings is presented in Table 1.

Broadcasts in DCGN had similar performance to that of sends. However, small-to-medium sized broadcasts with only CPUs in DCGN were actually faster than the corresponding MVAPICH2 broadcasts. This is because of the way DCGN handles collectives. If a process hosts more than one DCGN rank, the communication is dealt with internally via `memcpy` and `cudaMemcpy`. In MVAPICH2, inter-process communication (IPC) or Infiniband are used to perform the data copy.

As the size of a broadcast grew to several megabytes, the time for an all-CPU DCGN broadcast was equal to that of an MVAPICH2 broadcast. The DCGN GPU broadcast was still slower, again due to the overhead in performing communications with the GPU. A full graph of broadcast timings is shown in Figure 7.

Looking Forward These results show that DCGN performance is very close to an MPI implementation. DCGN can close the gap on or even overtake MPI's performance with vendor support. Several things are necessary: A method for signaling the CPU from the GPU, a direct connection to the NIC, a direct GPU-to-GPU connection via PCI-e, and buffers in system memory so the GPU may push data. We believe these additions would put DCGN on par with MPI while preserving its advantage of a higher-level, more flexible interface.

Nodes	Configuration	MPI (CPU)	DCGN (CPU)	Ratio	DCGN (GPU)	Ratio	DCGN (CPU+GPU)	Ratio
1	2 CPUs/0 GPUs	3 μ s	38 μ s	12.67 \times				
1	0 CPUs/2 GPUs				313 μ s	104.3 \times		
1	1 CPU/1 GPU						50 μ s	16.67 \times
1	2 CPUs/2 GPUs						53 μ s	10.60 \times
2	4 CPUs/0 GPUs	5 μ s	41 μ s	8.20 \times				
2	0 CPUs/4 GPUs				747 μ s	149.40 \times		
2	4 CPUs/4 GPUs						55 μ s	9.17 \times
4	8 CPUs/0 GPUs	6 μ s	43 μ s	7.17 \times				
4	0 CPUs/8 GPUs				806 μ s	134.33 \times		
4	8 CPUs/8 GPUs						70 μ s	—

Table 1: Barrier Timings for CPUs and GPUs. Inter-thread messages in DCGN add significant overhead, especially to barriers as no data is transferred. For DCGN applications that use GPUs, we compare the total number of kernels executing to MPI with an equal number of CPUs. These timings are not directly comparable as significantly more work is done to perform a barrier by a GPU than to perform a barrier by a CPU. However, these numbers provide a baseline for current and future comparisons.

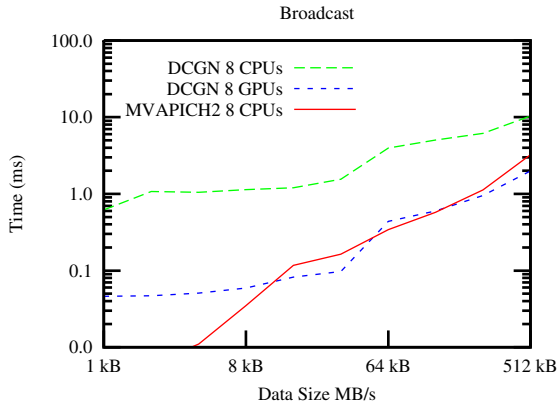


Figure 7: Broadcast timings with and without DCGN. The graph shows timings with eight DCGN ranks. Up to two MPI processes or DCGN-CPU threads run on the same node. As is expected, GPUs introduce slowdown. Much of this is attributed to the two trips over PCI-e for GPU communications. DCGN is faster in certain cases because the underlying MPI broadcast is executed with one-half as many participating ranks.

6. Discussion

6.1. Programming Model

Scientists want to take full advantage of the computational power of GPUs, Cells, and other DPMs. Allowing them the ability to develop kernels with full access to underlying hardware and software, and providing a message passing interface, is essential to giving scientists the access and power they desire. Other existing software packages either constrict developers and enforce a restrictive programming model, or provide access to raw resources with no higher-level support.

DCGN achieves an excellent trade-off in allowing full access to GPUs as DPMs while providing higher-level support. Developers use fully-functional kernels, and communicate through non-invasive techniques. Instead of forcing a user into treating their computation resources as slave devices with one very specific purpose, DCGN allows developers to treat GPUs as first-class computing resources with a wide range of power.

Very few tradeoffs are made in application development with DCGN. Users must implement kernels in a callback style fashion. This adds virtually no complexity to applications. At the absolute maximum, developers must define one or two additional structures, and these structures are only used to pass parameters to kernels via DCGN.

To say DCGN doesn't impose limitations on developers is false. On the CPU side, MPI is always off-limits, and threading libraries can only be used under certain conditions. As DCGN uses MPI, developers mustn't use MPI. Since communication is provided by DCGN, this restriction is actually moot; another communication library is not needed.

DCGN limits access to certain features of the GPU. DPMs map a number of data elements to threads. GPUs schedule blocks of threads to work, afterwards the blocks die. The blocks stay scheduled until completion, and it's common to use thousands to millions of blocks. DCGN requires persistent blocks to handle synchronous communication. The number of blocks can be reduced by employing a work queue instead of numerous blocks; work-queues help to ensure a balanced load.

6.2. Communication Patterns

DCGN provides dynamic communication but with a high cost. Polling creates a significant CPU load. Transferring GPU memory and synchronizing kernels with communications add overhead. Synchronization and tradeoffs in the implementation change the expected semantics of intra-

node communications to be globally synchronized. This reduces back-end complexity, but at a potential cost to performance. When intra-node communication occurs, the communication thread performs memory copies instead of using MPI. Local sends finish upon matching with a local receive and performing the memory copy. Local sends to a GPU will not have to wait for the memory to be copied to the GPU, but they'll wait for memory to be copied to a temporary CPU buffer.

This method was written to avoid using MPI for local communication. It was also written to make implementing the architecture of DCGN much easier. Communication requests that arise from DCGN all look the same to the MPI thread. All calls provide a communication partner (or root, in the case of most collectives) and potentially a buffer and byte-count. There is no distinction made by the DCGN thread between requests made by a CPU and requests made by a GPU. This allowed for rapid development and high level optimizations of DCGN.

7. Conclusions

Over time, as modern processors become more parallel and more heterogeneous, we submit that it will become increasingly important to build and support abstractions that allow both high performance and utilization of all system resources as well as permit many flexible and powerful programming models to be built atop those abstractions.

Creating a die with both a CPU and GPU, such as the upcoming AMD Fusion® and Intel Nehalem® processors, perfectly exemplifies this trend towards heterogeneity. As this trend continues, many believe that processors will have more components than can be turned on simultaneously due to power and heat restrictions.

The current methods for working with multi-GPU and heterogeneous systems are insufficient. Many problems exist that cannot be easily mapped onto a GAS or Global Arrays programming model. It is a hard question to answer: "How can we integrate parallel systems into a heterogeneous environment?"

DCGN enables us to merge heterogeneous computing resources and make a bigger, more powerful computing platform. At a high level, DCGN permits building scalable applications. One can write an application for a GPU, or for many hundreds of CPUs and GPUs and still expect scalable performance. Not only can scalable applications be built upon DCGN, but so can new and better programming models, as well as new systems. Developers can leverage the dynamic-communication capabilities to create virtually any communication model. DCGN provides a road map for future devices and drivers. Chipset and coprocessor manufacturers have a clear idea of the need for fast paths not just to the CPU, but to the NIC and other PCI-e siblings. Devices and drivers will hopefully support such

paths, allowing DCGN and other libraries' performance to rival that of CPU-based communication libraries, thereby eliminating all the limitations of DCGN.

It's important to understand that DCGN does all these things in the right manner. By separating the user-level interface from the lower-level communications, DCGN makes technological innovations immediately accessible to developers with no change to their code. This also allows device manufacturers and driver writers to test new hardware and systems software with benchmarks and open-source applications designed to test such systems.

References

- [1] AMD Corp. CrossFire. <http://ati.amd.com/technology/crossfire/features.html>, 2006.
- [2] NVIDIA Corp. NVIDIA Launches Revolutionary New Multi-GPU Technology. http://www.nvidia.com/object/IO_14172.html, 2004.
- [3] NVIDIA Corp. NVIDIA CUDA Compute Unified Device Architecture. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf, 2007.
- [4] NVIDIA Corp. Press Release: New NVIDIA Products Transform the PC Into the Definitive Gaming Platform. http://www.nvidia.com/object/IO_37234.html, 2008.
- [5] Z. Fan, F. Qiu, and A. E. Kaufman. Zippy: A framework for computation and visualization on a GPU cluster. *Computer Graphics Forum*, 27(2), June 2008.
- [6] H. Lee, J. P. Robertson, and J. A. B. Fortes. Generalized Cannon's algorithm for parallel matrix multiplication. In *ICS '97: Proceedings of the 11th International Conference on Supercomputing*, pages 44–51, New York, NY, 1997. ACM.
- [7] B. B. Mandelbrot and J. A. Wheeler. The Fractal Geometry of Nature. In *American Journal of Physics*, volume 51, pages 286–287, 1983.
- [8] Adam Moerschell and John D. Owens. Distributed texture memory in a multi-GPU environment. *Computer Graphics Forum*, 27(1):130–151, March 2008.
- [9] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A portable "shared-memory" programming model for distributed memory computers. In *Supercomputing '94: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, pages 340–349, New York, NY, 1994. ACM.
- [10] L. Nyland, M. Harris, and J. Prins. Fast N-Body Simulation with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, pages 677–695. Addison Wesley, 2007.
- [11] P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 1996.
- [12] M. Strengert, C. Miller, C. Dachsbacher, and T. Ertl. CUD-ASA: Compute Unified Device and Systems Architecture. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV08)*, pages 49–56, 2008.