IVM: A Task-based Shared Memory Programming Model and Runtime System to Enable Uniform Access to CPU-GPU Clusters

Kittisak Sajjapongse, Ruidong Gu, Michela Becchi

University of Missouri - Dept. of Electrical and Computer Engineering

{kittisak.sajjapongse,ruidong.gu}@mail.missouri.edu, becchim@missouri.edu

ABSTRACT

GPUs have been widely used to accelerate a variety of applications from different domains and have become part of high-performance computing clusters. Yet, the use of GPUs within distributed applications still faces significant challenges in terms of programmability and performance portability. The use of popular programming models for distributed applications (such as MPI, SHMEM, and Charm++) in combination with GPU programming frameworks (such as CUDA and OpenCL) exposes to the programmer disjoint memory address spaces and provides a non-uniform view of compute resources (i.e., CPUs and GPUs). In addition, these programming models often perform static assignment of tasks to compute resources and require significant programming effort to embed dynamic scheduling and load balancing mechanisms within the application.

In this work, we propose a programming framework called Inter-node Virtual Memory (IVM) that provides the programmer with a uniform view of compute resources and memory spaces within a CPU-GPU cluster, and a mechanism to easily incorporate load balancing within the application. We compare MPI, Charm++ and IVM on four distributed GPU applications. Our experimental results show that, while the main goal of IVM is programmer productivity, the use of the load balancing mechanisms offered by this framework can also lead to performance gains over existing frameworks.

1. INTRODUCTION

Thanks to their high computational power, their massive hardware parallelism and their increased programmability, Graphic Processing Units (GPUs) have been widely used to accelerate a variety of applications from different domains [1]. The popularity of GPUs is highlighted by their increased adoption in HPC clusters. Indeed, four of the top supercomputers in the world – Tianhe-2, Titan, Piz Daint and Stampede – are equipped with Nvidia GPUs. Yet, the use of GPUs within distributed applications still faces challenges in terms of programmability and performance optimization. In fact, despite recent integration efforts (e.g. CUDA-aware MPI [2]), the combined use of popular programming models for distributed applications (such as MPI, SHMEM, and Charm++ [3]) and for GPUs (such as CUDA and OpenCL) presents several issues and limitations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

CF'16, May 16-19, 2016, Como, Italy.

© 2016 ACM. ISBN 978-1-4503-4128-8/16/05...\$15.00

DOI: <http://dx.doi.org/10.1145/2903150.2903174>.

First, MPI, SHMEM and Charm++ expose disjoint address spaces to programmers. Since CPUs and GPUs have their own memory spaces, programmers need to explicitly manage data objects on different compute resources. In addition, MPI assumes a private memory space for each process. Furthermore, these programming models do not support the concept of shared objects: data objects belong entirely to a task (or process) in a distributed application and cannot be shared among tasks. Not only does this assumption complicate programmability, but it can also lead to inefficiencies in terms of memory utilization. For example, multiple tasks executing on the same GPU will need to duplicate even read-only data objects. Second, these programming models offer fairly basic support for GPUs. In particular, CPU and GPU resources are treated and accessed differently. Finally, hybrid high-performance computing clusters can present hardware heterogeneity at different granularities: besides consisting of nodes comprising both CPUs and GPUs, they can include nodes with different hardware configurations and GPUs with different compute capabilities even on a single node. Traditional programming models, such as MPI and SHMEM, offer static mechanisms to map work onto compute elements and require substantial programming effort to embed adaptive scheduling mechanisms within the application. This limits the performance portability of distributed GPU-accelerated applications. In addition, MPI and SHMEM require the use of cluster resource managers such as TORQUE and SLURM in order to provide inter-application load balancing. Unfortunately, these scheduling frameworks offer only basic support for heterogeneity. In order to provide load balancing in heterogeneous settings, these tools require the integration with recently proposed GPU virtualization frameworks [4-10]; however, these frameworks are still at a prototype phase. While Charm++ offers a mechanism to adaptively distribute load to compute resources, this load balancing mechanism is suited for computing environments with limited heterogeneity (for example, clusters including resources with similar compute capabilities), but may lead to performance limitations in more heterogeneous clusters.

In this work, we aim to overcome these limitations. To this end, we propose Inter-node Virtual Memory (IVM) – a flexible programming framework for distributed CPU-GPU applications. Our framework presents the following distinctive features. First, it reconciles memory spaces and provides the view of a *single virtual memory space* across nodes and heterogeneous compute resources. Second, it provides a *unified view of compute resources* and treats CPUs and GPUs similarly. Third, it supports *one-sided communication primitives*, which simplify the design of memory consistency policies. Fourth, it supports *dynamic task creation* (DTC), a mechanism to facilitate dynamic load balancing. Finally, our framework is designed for integration with cluster-level scheduling systems that enable inter-application scheduling and load balancing. We summarize our contributions as follows:

- We propose IVM – a novel programming framework that consists of a shared memory-based programming model and a

runtime system. IVM increases the programmability of heterogeneous distributed applications and offers easy-to-use support for dynamic load balancing.

- We propose two self-tuning load balancing schemes (*dynamic spawning* and *online monitoring*) based on our programming model and demonstrate their use on a variety of applications.
- We evaluate IVM on four distributed applications with different characteristics, and we compare it with MPI and Charm++. Besides increasing programmability, IVM improves the performance by up to a factor 2.5x and 1.85x over MPI-CUDA and Charm-CUDA, respectively.

2. BACKGROUND & RELATED WORK

We classify existing programming frameworks for distributed applications into two categories according to their execution model: *Cooperative-Process* (CP) and *Task-Based* (TB) frameworks. In the CP model, a computation is divided into parallel *processes* that execute for the entire application lifetime; processes work cooperatively and exchange information to collectively perform computation. In the TB model, *tasks* do not necessarily remain active for the entire application lifetime and often consist of small portions of code. TB frameworks usually use over-decomposition to break the computation into multiple tasks [11], and they allow programmers to describe parallel computation at a finer grain. Popular CP programming frameworks include the Message Passing Interface (MPI), Unified Parallel C (UPC) [12], Charm++ [3], and Symmetric Hierarchical Memory (SHMEM) [13]. TB programming frameworks include Legion [14], HPX [15], Chapel [16], X10 [17] and our proposed IVM. Although programming frameworks in each category share characteristics in the terms of execution, they differ in other important aspects, as we discuss below.

2.1 Memory Model

Henceforth, we use the term “task” to refer to both processes and tasks in the CP and TB models. The main challenge in the design of a memory model for distributed memory architectures (e.g. CPU-GPU clusters) lies in the existence of disjoint physical address spaces. Because each compute resource (i.e. CPU, GPU) and node can have its own physical memory space, disjoint memory spaces can exist both within and across compute nodes. Two well-known memory models, message passing and shared memory, have been proposed and extensively studied to hide the complexity of disjoint memory spaces from the programmer.

MPI employs the message-passing model, in which tasks have their own private space and do not share data structures. Message passing exposes the physical memory hierarchy of the cluster to the programmer and relies on her to properly manage the placement of data on compute resources. MPI communication and synchronization primitives are based on the two-sided communication paradigm: *send* and *receive* primitives are used to exchange data, and collective primitives such as *barriers* are used for synchronization. These primitives require participation of all the parties that need to communicate or synchronize. The overlapping of computation and communication requires substantial programming effort. Some extensions such as Unified Virtual Addressing (UVA) [18], Shared Virtual Memory (SVM) [19], and CUDA-aware MPI [2], have been proposed to hide the disjoint memory space complexity; however, they do so only at the intra-node level. MPI-2 offers also one-sided communication primitives, that we discuss in Section 5.4.

UPC, Charm++¹, SHMEM, Legion, HPX, Chapel, X10 and our proposed IVM employ the shared memory model. This model provides the abstraction of a shared virtual address space over physically disjoint memory spaces, thus simplifying programming. Communication primitives are in this case based on the one-sided communication paradigm, do not require all communication parties to actively participate, and allow tasks to write and read information from a specific location in the shared virtual address space without being aware of its physical location. Although all of these frameworks address the issue of disjoint memory spaces at the inter-node level, only Legion and IVM fully address this issue at the intra-node level.

Synchronizations—whether implicit or explicit—are required to ensure data dependencies are upheld for proper program execution. SHMEM, UPC, MPI, Chapel, and X10 allow explicit synchronization through barriers and synchronization variables. Charm++ uses implicit synchronization. Likewise, Legion and HPX primarily rely on implicit synchronization. In particular, they allow programmers to describe data dependencies by specifying the access types, privileges, and data-locality of tasks and let then the runtime system synchronize and schedule tasks transparently. IVM supports both kinds of synchronization: it provides explicit synchronization primitives on the shared memory space and implicit synchronization through the DTC mechanism.

Most of these programming models offer only basic memory allocation and data sharing mechanisms. For example, SHMEM assumes that memory allocations are performed in a collective manner and the shared memory space is visible to all tasks. Legion assumes that a head node is capable of holding all the data used by the application; the runtime system then distributes portions of data and tasks to nodes transparently. Charm++ requires users to initialize the data in an object-oriented manner at the beginning. These mechanisms, however, do not allow tasks to selectively allocate or share data structures, lacking in flexibility. In addition, by not taking advantage of the aggregated memory capacity of a cluster, they suffer from lack of scalability. HPX and PGAS-based frameworks (UPC, X10 and Chapel) allow data to be distributed across physical memories, but require users to specify their distribution strategy.

Similarly to PGAS-based frameworks, IVM allows data distribution. Specifically, in IVM none of the tasks has ownership over data structures. While tasks can allocate data, when they terminate they can leave these data in the shared space thus allowing other tasks to gain access to them. In addition, IVM offers a mechanism to improve data locality (Section 3). With the assistance of the DTC mechanism, tasks can flexibly allocate and share data structures. While other frameworks, such as UPC, have similar memory allocation and access semantics, their execution model is different (CP model) and does not offer mechanisms to flexibly create and manage tasks.

2.2 Load Balancing

Since in the CP model tasks are created at initialization and have the lifetime of the application, CP frameworks support load balancing either through task migration (Charm++, UPC) or by explicitly transferring load among tasks using communication primitives (MPI, SHMEM) [20-22]. Task migration in Charm++ is performed transparently by the runtime system based on the performance capabilities of the available compute resources and

¹ We categorize Charm++ into the shared model since it provides an object migration abstraction.

Figure 1 – IVM’s Execution and Memory Models

on online profiling. Because in Charm++ data objects must be migrated along with tasks, task migration incurs a data transfer overhead. MPI-2 defines a mechanism to spawn tasks dynamically for load balancing purposes. The use of this feature, however, requires non-trivial coding effort: because parent and child tasks belong to different MPI groups, the user is left with the burden of using MPI intra- and inter-communicators.

The TB model supports dynamic task creation and termination, thus allowing the easy implementation of scheduling and load balancing schemes either in the runtime system or in the application. In Legion and HPX load balancing is performed by the runtime system. Specifically, Legion handles load balancing in a centralized fashion within the head node, leading to scalability issues. In HPX the dataset is distributed across compute resources, and tasks are then spawned on compute resources with the required portions of dataset. X10 and Chapel do not directly implement load balancing in the runtime, but provide primitives and mechanisms to embed it in the application. Similarly to X10 and Chapel, IVM provides the programmer with the flexibility to easily embed custom load balancing schemes in the application. This is possible through DTC, which allows tasks running on a resource to spawn new tasks on a different or the same resource and offload computation to them. Additionally, it is possible to embed a scheduling component in the IVM runtime and delegate to it the mapping of the new tasks to resources.

2.3 Programming Languages versus Runtime Libraries

While MPI, HPX, Legion and SHMEM are available as runtime libraries that can be invoked from C programs, Chapel and X10 are full-blown programming languages that are compiled into C, C++, and Java. As a consequence, the widespread adoption of these two systems is hampered by the need for the programmer to learn and understand their syntax, semantics and execution model, and by the added debugging complexity. Being an extension of C and C++, Charm++ and UPC are more easily adopted. To simplify its use, we implement IVM as a C runtime library. In addition, given the widespread adoption of MPI, we use initialization, finalization and task handling primitives similar to those of this standard, while deviating from it in the data and resource management primitives.

2.4 GPU Support

All the considered frameworks provide basic GPU support. CUDA-aware MPI simplifies the handling of CPU-GPU data transfers from MPI code. Charm++, SHMEM, HPX and UPC provide wrapper functions for users to enqueue GPU work and perform CPU-GPU data transfers implicitly, facilitating the integration of CUDA code. Some of these systems offer additional features. SHMEM relies on CUDA UVA to simplify memory

handling (this feature, however, is available only on some GPU devices). Charm++ supports pipelined execution of GPU code. UPC enables the shared heap management of GPU device memory. Chapel and X10 compilers have been extended with CUDA code generation capabilities [23]. While Legion provides some form of GPU abstraction, the other frameworks require users to be aware of the cluster setup and in some cases to explicitly select the GPUs to be used. IVM, on the other hand, aims to provide uniform access to CPU and GPU resources. To this end, it provides full GPU abstraction: besides hiding the location of GPUs in the cluster and the CPU-GPU data transfers, it automatically schedules code on GPUs, provides full memory utilization, and enables GPUs to be shared by applications.

2.5 Summary of IVM design goals

In summary, our goal is to provide a task-based runtime library that improves programmability of CPU-GPU clusters. Our IVM design aims for a balance between abstraction and flexibility. IVM also offers GPU virtualization, thus allowing homogeneous access to CPU-GPU. Further, it includes an easy-to-use dynamic task creation mechanism to easily embed load balancing in the application. IVM can also be considered a *substrate* on top of which it is possible to build programming frameworks offering a higher degree of abstraction (for example, automatic data partitioning and task spawning mechanisms).

3. IVM FRAMEWORK DESIGN

In this section, we describe our IVM framework in more detail. We start by discussing IVM’s execution and memory models. Next, we present our system design, including IVM’s software components. Finally, we illustrate IVM’s programming interface.

3.1 Execution Model

We recall that the main goal of the IVM framework is to increase the programmability of applications running on distributed memory systems that include CPUs and GPUs. This is done by providing a uniform view of memory spaces and compute resources. Specifically, IVM adopts a shared memory model with relaxed memory consistency; the programmer sees a single virtual memory space across all compute resources – CPUs and GPUs. Since all compute resources have access to data objects allocated in this virtual memory space, the programmer can access CPUs and GPUs in a unified way; in the IVM programming interface, all compute resources are represented by *device* instances.

Under IVM, an application consists of multiple Processing Elements (PEs). In essence, a PE is a task that is bound to a specific compute resource. Figure 1 shows the physical view of a compute cluster consisting of heterogeneous nodes and including physical memory spaces (to the bottom), and its abstract view (to the top). As can be seen, IVM provides the view of a virtual memory space containing shared data objects accessed by PEs.

Similarly to MPI, a distributed IVM application is started by executing its binaries. Upon instantiation the IVM framework will create a single root-PE, which is the ancestor of all PEs belonging to the application. The root-PE can create its immediate child-PEs and distribute the work to them. It is possible for the child-PEs to further spawn additional PEs. In general, the framework allows a PE to spawn one or more PEs, which can bind to the same or different devices. The IVM framework also provides a mechanism to synchronize PEs. This mechanism consists of wait and signal primitives and is similar to POSIX-threads’ conditional variables. For example, a PE can wait for its children by invoking the wait primitive and specifying the set of PEs to wait on; the child-PEs

can then invoke the signal primitive to notify the waiting PE to continue. Programmers can also implement their own synchronization mechanisms using IVM's shared memory. For example, busy-waiting, in which a PE waits on a variable written by another PE, can be implemented using IVM's synchronization primitives discussed in Section 3.5.

3.2 Memory Model

Figure 1 shows shared data objects accessed by PEs through the virtual memory space. A data object can be allocated by one and only one PE. Other PEs can gain access to the data object through a mapping operation. PEs can allocate objects at different times and selectively share the objects with a group of PEs. This property enables applications to define their data sharing schemes and to optimize the cluster-wide memory usage and data locality. Upon allocation, each data object is associated with an identifier that can be used by the IVM framework to reference that object. Different PEs can access a data object by providing its identifier to the IVM runtime through the programming interface. Note that data objects allocated in the virtual space do not belong to any specific PEs. All PEs will have the illusion that the data objects were floating singletons residing in the virtual space.

On the top part of Figure 1 the virtual memory space contains two data objects which can be accessed by the PEs. However, the framework needs to manage the physical data corresponding to each virtual data object. IVM manages the data objects by transparently allocating memory for them on different physical spaces. As can be seen in the bottom part of Figure 1, there may be multiple physical copies of each (virtual) data object, and those may reside on different devices. If a data object is accessed by multiple GPUs, each of these GPUs will require a copy of the data object in its physical memory. Physical copies of data objects can be of two kinds: *master-copies* and *mirror-copies*. A master-copy is the copy of the data object that is initially allocated by the instantiating PE, whereas a mirror-copy is a copy that is created upon a mapping operation. Mirror-copies are created only if the allocating and the mapping PE are bound to different devices or physical memory spaces. If all PEs are associated to the same device, only the master-copy will be allocated. These physical copies are not exposed to the programmer, who sees only the data objects residing in the virtual space.

The IVM framework synchronizes the master- and mirror-copies of data objects through the *put* and *get* primitives. When a PE invokes a put/get operation on a data object, the entire or part of the master- and mirror-copies of the data object are synchronized. The put/get primitives will have no effect for PEs that share the same physical copy. Beside simple put/get primitives, *gather* and *scatter* operations allow for a master-copy to synchronize with different mirror-copies and for a master-copy to distribute contents to different mirror-copies, respectively.

On a node, the framework allocates master- and mirror-copies within Linux's shared memory. Therefore, data consistency among PEs sharing the same copy is strict. This eliminates the need for duplicating data objects for PEs working on the same device or physical memory space and may reduce the memory footprint of the application. The allocation method on GPUs is different and will be described in Section 3.4.

3.3 System Design

The IVM framework consists of two main components: the runtime library and the runtime daemon. The IVM runtime library implements the programming API functions described in Section 3.5 and listed in Table 1. The runtime daemon runs on each

Figure 2 – System design and GPU support

compute node and serves API requests issued by applications. These requests include: PE registration/deregistration, memory allocation/de-allocation, memory put/get, and PE creation and destruction.

Figure 2(a) shows the overall design of the IVM framework. As can be seen, a single instance of the runtime daemon runs on each compute node and serves requests from the PEs physically residing on that node. Daemons running on different compute nodes communicate through inter-node communication channels which support Ethernet networks and Infiniband fabrics. Programmers can specifically choose preferred physical links to be used for data objects synchronization purposes. For simplicity, we implement inter-node communication using a master/slave model whereby a single root-daemon services requests issued by slave-daemons.

As mentioned above, a user starts an IVM application by executing the application's binaries. Once the operating system has created a task (process) for the application, the task will register itself to the daemon running on the local node. The task and local daemon will be considered the root-PE and root-daemon for the application. The root-PE can request memory allocation, memory mapping, PE-creation, and PE-destruction. Communication between PEs and their associated daemon are performed through Linux Inter-Process-Communication (IPC) mechanisms. PEs send requests to the daemon by invoking the API functions described in Section 3.5. If the servicing daemon is a slave-daemon, it will determine from the type of request whether it can be serviced locally without the assistance of the root-daemon. If the request needs assistance from the root-daemon, the slave-daemon will forward the request to it and wait for a response. The local daemon notifies the PEs upon completion of their requests.

When a daemon receives allocation requests, it allocates the master-copy of the data object using Linux shared memory, and it returns the object's reference to the requesting PE. At this point, the object is immediately available to all PEs that reside on the same physical node. However, the object is not allocated on other compute nodes until one of the remote PEs issues a mapping request for the object. Once the first memory-map request for a data object is received by a remote daemon, this daemon allocates a mirror-copy of the data object, synchronizes the content of the mirror-copy with the master-copy, and returns the reference of the mirror-copy to the requesting PE. The root-daemon provides a directory service for mapping operations. Hence, master-copies can be distributed to different physical memory spaces. Subsequent mapping requests of the same data object will not incur memory allocation or synchronization. Further synchronization between data copies requires the use of *put* and *get* API primitives.

Table 1: IVM Programming API

API Functions	Description
Initialization & Finalization	
<i>ivmEnter()</i>	Registers a PE - This function should be the first IVM call.
<i>ivmExit()</i>	Unregisters a PE - This function should be the last IVM call.
Device Management	
<i>ivmCreateDev()</i>	Creates a device instance to represent a CPU/GPU.
<i>ivmDestroyDev()</i>	Releases resources allocated for a device instance.
Processing Element Management	
<i>ivmCreatePEs()</i>	Spawns one or more PEs on a specified device.
<i>ivmKillPEs()</i>	Terminates one or more PEs on a specified device.
<i>ivmGetMyId()</i>	Retrieves identification number of the calling PE.
<i>ivmGetMyGroupId()</i>	Retrieves group identification number of the calling PE.
<i>ivmGetMyDevType()</i>	Retrieves type of the device to which the calling PE is bound.
<i>ivmSetMaxPesId()</i>	Sets the maximum identification number of PEs that can be spawned. Subsequent calls to <i>ivmCreatePes()</i> after the maximum identification number is reached have no effect.
<i>ivmWait()</i>	Waits for a specified PE.
<i>ivmSignal()</i>	Signals the waiting PE to continue.
Memory Management	
<i>ivmMalloc()</i>	Allocates memory in the virtual memory space.
<i>ivmFree()</i>	Releases memory in the virtual memory space.
<i>ivmMap()</i>	Maps to a memory object in the virtual memory space.
<i>ivmMapSubset()</i>	Maps a specific region of a memory object.
<i>ivmUnmap()</i>	Un-maps from a memory object in the virtual memory space.
<i>ivmSyncPut()</i>	Writes content from the specified reference to the specified reference in the master-copy.
<i>ivmSyncGet()</i>	Reads content from the specified reference in the master-copy to the specified reference in a mirror-copy.
<i>ivmSyncPutGroup()</i>	Writes content from a list of references in the master-copy to a list of references in mirror-copies (Scatter/Broadcast).
<i>ivmSyncGetGroup()</i>	Reads content from a list of references in mirror-copies to a list of references in the master-copy (Gather).

3.4 GPU Support

In the IVM framework, PEs are allowed to share physical data objects (master-/mirror-copies). However, commonly used GPU software stacks (e.g., CUDA and OpenCL runtimes) offer limited support for sharing GPU memory across tasks. In particular, the CUDA runtime associates a different memory address space to each task that uses a GPU. Therefore, a naïve design would allocate on GPU multiple data copies of each shared data objects – one per PE. This solution would be inefficient in terms of memory usage, especially when the data objects are read-only. In addition, it would lead to multiple and unnecessary GPU memory allocations and CPU-GPU data transfers. To address this issue, we include in our IVM framework one additional component: the *IVM GPU-daemon*. This daemon, shown in Figure 2(b), follows the design of the GPU virtualization runtime system described in [8]. Specifically, the IVM GPU-daemon consists of a frontend library and a backend daemon. The frontend library intercepts CUDA calls and redirects them to the backend daemon, which decides which requests should be issued to the CUDA runtime. In case of IVM, CUDA calls are generated only by the IVM daemon. In fact, applications have a uniform interface to CPUs and GPUs, and access their memories only through IVM API primitives.

As shown in Figure 2(b), the IVM runtime includes two memory paths: one for PEs associated to CPUs, and the other for PEs associated to GPUs. The PEs executing on CPU perform memory allocation and mapping through the IVM daemon. The memory-related operations originating from PEs executing on the GPU go first through the IVM daemon and then through the IVM GPU daemon. When a PE performs a memory operation, the IVM daemon determines the compute resource used by the PE. If the resource is a GPU, then the IVM daemon generates the required CUDA calls to complete the operations. These CUDA calls are intercepted by the frontend library and redirected to the backend GPU daemon. By controlling all memory operations issued to GPUs, the GPU-daemon can avoid multiple physical copies of

```

1: #define VEC_SIZE 5000
2: int id, gid;
3: int * vecA, * vecB, * vecC;
4: int vec_size = VEC_SIZE * sizeof(int);
5: int main(int argc, char ** argv) {
6:     ivmEnter();
7:     ivmGetMyId(&id); ivmGetMyGroupId(&gid);
8:     if (id == 0) {
9:         ivm_device d[2]; ivm_pe pe[2];
10:        ivmMalloc(&vecA, "VECA", vec_size);
11:        ivmMalloc(&vecB, "VECB", vec_size);
12:        ivmMalloc(&vecC, "VECC", vec_size);
13:        InitializeVectors();
14:        ivmCreateDev(host_0, IVM_CPU, &d[0]);
15:        ivmCreateDev(host_1, IVM_GPU_1, &d[1]);
16:        ivmCreatePEs(d[0], 1, &pe[0]);
17:        ivmCreatePEs(d[1], 1, &pe[1]);
18:        ivmWait(pe[0]);
19:        ivmWait(pe[1]);
20:    } else {
21:        int start_pos = (id - 1) * 2500;
22:        int end_pos = start_pos + 2500;
23:        ivmMap(&vecA, "VECA", 2500);
24:        ivmMap(&vecB, "VECB", 2500);
25:        ivmMap(&vecC, "VECC", 2500);
26:        vectorAdd(start_pos, end_pos);
27:        ivmSyncPut(&vecC[start_pos], vec_size);
28:        ivmSignal();
29:    } ivmExit(); }

```

Listing 1: Example of use of IVM API: vector addition

data objects shared by PEs mapped to the same GPU. This can be done by keeping track of the identifiers of the data objects allocated on each GPU, and selectively ignoring any subsequent *cudaMalloc* associated to the same data object. In summary, this design bypasses the restrictions of the CUDA runtime and allows multiple PEs using the same GPUs to share data objects.

3.5 IVM Programming Interface

The IVM API, shown in Table 1, consists of four categories of primitives: *Initialization & Finalization*, *Device Management*, *PE Management* and *Memory Management*.

The *ivmEnter* and *ivmExit* primitives belonging to the first category allow the programmer to register and deregister PEs with the IVM daemon, and their use is similar to that of MPI's *MPI_Init* and *MPI_Finalize*. These functions do not cause inter-PE communication or synchronization. PEs belonging to an application can invoke these functions asynchronously even after Dynamic Task Creation.

The device management primitives are used to create and destroy instances of *devices* – either CPUs or GPUs – and allow uniform access to them.

The PE management primitives allow the dynamic creation and release of PEs. Upon creation, each PE is associated to a specific device and assigned an identification number and a group identification number by the IVM runtime. The group identifier is meant for use by scheduling frameworks implemented on top of the IVM layer. The *ivmWait* and *ivmSignal* primitives facilitate synchronization among PEs and offer a mechanism similar to POSIX threads' conditional variables.

The memory management functions allow creating data objects within the shared virtual memory space and managing the consistency between copies of these objects residing on different physical memory spaces. Each data object is created by a PE through the *ivmMalloc* primitive and can be accessed by other PEs by invoking the *ivmMap* and *ivmMapSubset* memory mapping functions. The former allows a PE to access an object in its entirety whereas the latter allows a PE to access only a contiguous portion of an object and is intended to reduce data transfers between compute nodes and devices. We recall that each PE is spawned on a specific device; the IVM runtime uses the

```

1 int main(int argc, char ** argv) {
2     ivmEnter();
3     Get my ID and my group ID
4     int *sync;
5     if (id == 0) { /* Root-PE */
6         ivmMalloc(&sync, "sync", ... );
7         ivmMalloc(results, "results", ...);
8         Create devices
9         for (i=0;i<ITERATIONS;i++) {
10             Set all slots of sync to 0
11             ivmSetMaxPesId(MAX_PES * (i+1));
12             Create one PE on each device
13             Wait until all slots of sync are 1
14         }
15     } else { /* Child-PEs */
16         ivmMap(&sync, "sync", ... );
17         ivmMap(&results, "results", ...);
18         int work_id = GetWork(my_id);
19         Perform computation
20         if (unassigned work items exist){
21             ivmCreateDev(&d,THIS_HOST,THIS_DEV);
22             ivmCreatePEs(d, 1);
23         }
24         ivmSyncPut(my results);
25         sync[my_id-1] = 1;
26         ivmSyncPut(&sync[my_id-1], ...);
27     }
28     ivmExit();

```

(a) Dynamic Spawning (DS-LB)

```

1 int main(int argc, char ** argv) {
2     ivmEnter();
3     Get my ID and my group ID
4     double *time; bool *done;
5     if (id == 0) { /* Root-PE */
6         ivmMalloc(&time, "time", ...);
7         ivmMalloc(done, "done", ...); *done = 0;
8         ivmMalloc(results, "results", ...);
9         Create devices
10        Create one PE on each device
11        for (i=0;i<ITERATIONS;i++) {
12            Wait all PEs to fill time
13            foreach (time[j])
14                if (time[j]/min(time) < threshold)
15                    Create PE on device[j]
16        } *done=1; ivmSyncPutGroup(done, ...);
17    } else { /* Child-PEs */
18        ivmMap(&time, "time", ...); ivmMap(&done, "done", ...);
19        ivmMap(&results, "results", ...);
20        int work_id = GetWork(my_id);
21        do {
22            if (# of total PEs has changed)
23                work_id = GetWork(my_id);
24            Perform and time computation
25            time[my_id-1] = reported duration;
26            ivmSyncPut(my results);
27            ivmSyncPut(&time[my_id-1], ...);
28        } while(*done == 0);
29    } ivmExit();

```

(b) Online Monitoring (OM-LB)

Listing 2: Pseudo-code for load balancing schemes

association between PEs and devices to determine the physical memory where the master-copy of each data object should reside. The put and get primitives allow managing the content of the data objects. Specifically, *ivmSyncGet* and *ivmSyncGetGroup* read and gather data from the master-copy of a data object to its mirror-copy, while *ivmSyncPut* and *ivmSyncPutGroup* write and scatter data from the mirror-copy of the data object to its master-copy. Note that it is the programmer’s responsibility to properly handle concurrent accesses to data objects by using these primitives; the IVM runtime does not handle consistency issues implicitly. All GPU-related memory operations generate corresponding CUDA function calls to the GPU-daemon. References to objects returned by the memory-related primitives can be used directly in GPU kernel calls.

Listing 1 illustrates the use of the IVM API to implement vector addition. The code assumes that the application is initially invoked using a single PE (the root-PE) and dynamically spawns two additional PEs: one on CPU and one on GPU. The application invokes *ivmEnter* and *ivmExit* to register and unregister PEs (lines 6 and 29). Similar to MPI tasks, after registration the PEs retrieve the identification numbers associated to them by the IVM runtime (line 7). Before starting the computation, the root-PE allocates the required arrays using *ivmMalloc* (lines 10-12). These arrays are visible to the devices (CPUs and GPUs) on which the PEs invoke *ivmMap*; their identifiers *VECA*, *VECB* and *VECC* can be used by the other PEs to get access to them. At lines 14-15 the root-PE creates instances of devices to represent the CPU and GPU on which the computation will be performed (*IVM_CPU* and *IVM_GPU_1*, respectively). It then offloads the computation to the child-PEs (lines 16-17) and waits for them to complete (lines 18-19). The execution of the child-PEs also starts from line 5. Each child-PE uses its identification number (obtained at line 7) to identify the portion of the arrays on which it needs to operate (lines 21-22). The mapping operations at lines 23-25 allow accessing the vectors allocated by the root-PE; if the child-PE does not execute on the same compute node or device as the root-PE, the mapping operations cause the automatic creation of a mirror-copy of the arrays. Note that this uniform interface frees the programmer from the need to account for the physical location of the variables and the nature of the devices performing the

computation. After the computation (line 26), the child-PEs write the results back to the master-copy via *ivmSyncPut* (line 27) and notify the root-PE to continue (line 28). The code assumes the availability of two versions of the *vectorAdd* function at line 26: one for CPU and one for GPU. The runtime uses the PE-to-device association to invoke the proper implementation.

4. LOAD BALANCING & APPLICATIONS

In this section we describe our proposed load balancing schemes and four benchmark applications that we have implemented with the IVM framework.

4.1 Load Balancing Schemes

In this section, we propose two load balancing schemes based on the DTC mechanism and show how they can be easily implemented using IVM API.

Dynamic Spawning Load Balancing (DS-LB) – The DS-LB scheme enables work to be dynamically assigned to resources. Figure 3(a) presents a graphical representation of DS-LB. At a high level, this mechanism works as follows. The overall computation is broken into “work-portions”, each executed by a PE. Each device will run a single PE at a time. When a PE finishes executing the work portion assigned to it, it spawns another PE on the same device. Because PEs associated to faster devices will spawn more PEs, more powerful devices will be assigned more work. In Figure 3(a), the solid and pitted arrows indicate the PEs spawned by the root- and child-PEs, respectively.

Listing 2(a) shows the pseudo-code for DS-LB. IVM API calls are bolded. Lines 6-13 are executed by the root-PE, while lines 16-26 are executed by the child-PEs (either spawned by the root-PE or by other child-PEs). Variable *sync* allows synchronization between root- and child-PEs. This variable, initialized by the root-PE (lines 6 and 10), contains a flag for each child-PE. The computation is broken in iterations (line 9), and up to *MAX_PES* PEs are spawned in each iteration (set at line 11). In each iteration, the root-PE first spawns a child-PE on each device (line 12) and waits for all PEs to complete (line 13). The child-PEs use their identification number to retrieve the work-portions that they must execute, and perform the computation (lines 18-19). Upon completion, if more work is still pending, each child-PE spawns a

Figure 3: Graphical representation of load balancing schemes

new PE on the same device where it resides (lines 20-23). Finally, every child-PE synchronizes its result with the master-copy (line 24), notifies the root-PE through the *sync* variable (lines 25-26), and terminates (line 28).

Online Monitoring Load Balancing (OM-LB) – At the high level, the OM-LB scheme performs load balancing by monitoring the performance of different resources and dynamically assigning more work to more powerful devices. Figure 3(b) illustrates the OM-LB scheme. The root-PE first distributes the work-portions equally to all resources as indicated by the solid arrows. During execution, the root-PE observes the performances of all PEs and spawns more PEs on the resources that can handle more work-portions. Once more PEs are created, the work-portions will be dynamically redistributed to allow the new PEs to participate in the computation. Differently from DS-LB, in this case all spawned PEs remain active for the entire computation, and each device can be time-shared by the executing PEs.

Listing 2(b) shows the pseudo-code of OM-LB. Lines 6-16 are executed by the root-PE, while lines 18-28 are executed by the child-PEs. The root-PE first allocates memory for variables *time* and *done* and for other data objects (lines 6-8). Variable *time* is an array used for synchronization purposes and holds the computation times reported back by the child-PEs. Variable *done* is used to notify the child-PEs when the entire computation is completed. The root-PE then creates devices, spawns PEs and waits for all PEs to report back (lines 9-10 and 12). Once all PEs have reported the time, the root-PE determines the performance differences among the PEs and spawns more work on the fastest devices (line 13-16). The child-PEs map to data objects and perform work division (lines 18-20). Child-PEs that have already worked in previous iterations determine whether the total number of PEs is changed and re-divide the work-portions as necessary (line 22-23). Finally, all the PEs write back the results and report the time (line 25-27). The PEs terminate when the root-PE indicates that the entire computation is completed through the *done* variable (lines 16 and 28).

4.2 Benchmark Applications

N-body simulation (NBODY) – NBODY is a simulation of a dynamical system of particles. The computation is performed in time-steps. In each time-step, attributes of all particles, i.e. position and velocity, are updated. Our IVM implementation of NBODY uses the DS-LB scheme. At each time-step, the calculation of particle attributes is divided into a number of work-portions, each containing a subset of particles. The child-PEs retrieve work-portions based on their identifiers and spawn child-PEs dynamically until the overall computation completes. The MPI-CUDA is very similar to the IVM version except that subsets of particles are statically assigned to tasks.

Table 2: Hardware setup

Node type	# of nodes	Attr.	Values
Type 1	2	CPU	20-core Intel Xeon@ E5@2.60 GHz, 128 GB
		GPU	Nvidia Tesla K40m (Kepler) 2,880 cores @ 745 MHz 12 GB memory @ 3000 MHz
Type 2	2	CPU	20-core Intel Xeon@ E5@2.60 GHz, 128 GB
		GPU	Nvidia Geforce GTX 480 (Fermi) 480-cores @ 1400 MHz 1.5 GB memory @ 1848 MHz
Type 3	6	CPU	20-core Intel Xeon@ E5@2.60 GHz, 128 GB
		GPU	Nvidia Tesla K20Xm (Kepler) 2,688 cores @ 732 MHz 6GB memory @ 2600 MHz

Dense Matrix Multiplication (DMM) – DMM multiplies two square matrices. Our IVM implementation of DMM uses the DS-LB scheme. We divide the result matrix into $M \times M$ work-portions along both dimensions. The root-PE initializes the matrices and spawns the child-PEs; the child-PEs can then spawn more PEs as necessary. Because DMM involves large data transfers, we use *ivmMapSubset* for the child-PEs to map only the required parts of the multiplicand and multiplier matrices. This allows breaking a single large transfer into multiple smaller transfers, overlapping computations and data transfers, and avoiding broadcasting the whole matrices to all PEs. The tiles that have already been mapped can be reused by other child-PEs on the same physical node. Our MPI-CUDA implementation is similar to the IVM version except that tasks are statically assigned portions of the result matrix column-wise and progress down the matrix.

Needleman-Wunsch (NW) – NW is a dynamic programming algorithm widely used in bioinformatics for comparing biological sequences [21]. Our IVM implementation of NW uses DS-LB. The root-PE and the child-PEs perform allocation and mapping of the entire dataset, respectively. Sequence-pairs are divided into multiple work-portions which can be dynamically retrieved by the child-PEs. In the MPI-CUDA version, the sequence-pairs are distributed to tasks equally.

Himeno (HIMENO) – HIMENO is a well-known benchmark application which implements a computational kernel found in the simulation of incompressible fluids. This kernel performs stencil computations on a 3-D grid of pressure values. We divide the computation in work-portions along the Z-axis of the grid. The MPI-CUDA version assigns to all tasks the same amount of work along the Z-axis. In the IVM implementation we enable point-to-point communication for exchanging XY-planes by allocating a shared buffer. The sending PEs write into the corresponding slots of the buffer and synchronize the contents with the master-copy. Then, the content of the master-copy is distributed to mirror-copies to allow the receiving PEs to access the corresponding slots. As the computation proceeds through iterations, the OM-LB scheme keeps performance records of all PEs and spawns more PEs on faster resources.

5. EXPERIMENTAL EVALUATION

In this section we evaluate the performance and effectiveness of the two proposed load balancing schemes using the benchmark applications described in Section 4.2. Our experiments cover two aspects: (i) a comparison between the applications described above implemented on top of the IVM and two widely used frameworks: MPI-CUDA and Charm-CUDA; and (ii) an analysis of the effect of different parameter settings on load distribution.

5.1 Experimental Setup

We conducted our experiments on a ten-node cluster that includes

(a) Speedup over baseline MPI-CUDA setting (10 processes for NBODY and NW, 9 processes for DMM). The number on the first bar indicates the execution time of the baseline MPI-CUDA code.

(b) Relative load distribution across node types. We recall that the system includes two *type-1* nodes, two *type-2* nodes, and six *type-3* nodes. Each bar represents the average load distribution for the nodes of any given type (in percentage terms).

Figure 4: Comparison between MPI-CUDA, Charm-CUDA and IVM (DS-LB policy)

ten Nvidia GPUs (one per node). Table 2 shows the hardware configuration of the compute nodes. As can be seen, the cluster includes three types of nodes that differ in the compute capability of their GPU (i.e. number of cores, memory and core speed). This hardware heterogeneity can cause load imbalance within applications. The ten nodes are interconnected through a 10G Ethernet. CentOS 7.1, OpenMPI 1.6.4 and CUDA 7.0 are installed on every node.

5.2 Dynamic Spawning Load Balancing

Performance Comparison – Figure 4(a) shows a performance comparison between MPI-CUDA, MPI-CUDA with dynamic load-balancing (MPI-CDYN), Charm-CUDA, and IVM using a varying number of tasks, chares and PEs. Hereafter, when we can do so without lack of clarity, we will use the abbreviation “PEs” also to refer to MPI tasks or Charm’s chares. The datasets used in these experiments have the following sizes: 1.2 million particles for NBODY, matrices with 10,000 x 10,000 elements for DMM, and 5,000 sequence pairs for NW. For NBODY and NW, we vary the number of PEs from 10 (the number of nodes and GPUs in the cluster) to 64. For DMM, we vary the number of PEs from 9 to 64 because we divide the work in the resulting matrix equally in both dimensions. In case of MPI-CUDA, the tasks are statically assigned to the GPUs in a round-robin fashion (that is, no load balancing takes place). In case of MPI-CDYN, tasks are dynamically created using the dynamic process creation provided in MPI-2. In case of Charm-CUDA, we enable the *refineLB* load-balancing scheme provided by the Charm++ framework. In case of IVM, we use the DS-LB scheme. Specifically, we divide the computation into a number of work-portions equal to the maximum number of PEs that we want to spawn over the execution of the application. Each PE will handle a work-portion. Initially the root-PE spawns 10 PEs in case of NBODY and NW, and 9 PEs in case of DMM; child-PEs are then spawned dynamically as needed until the number of PEs equals that of work-portions. The MPI-CDYN adopts the same load balancing

scheme as IVM. For NBODY and NW, our baseline is the performance of the 10-task configuration under MPI-CUDA and similarly for DMM the 9-task configuration.

Figure 4(a) shows the speedup/slowdown of all implementations over the baseline MPI-CUDA implementation as the number of PEs varies. When the number of PEs equals that of GPUs used (10 PEs for NBODY and NW, 9 PEs for DMM), MPI-CDYN, Charm-CUDA, and IVM do not perform any load balancing. As can be seen, in this situation all implementations of all applications achieve similar performance as the baseline. While improving programmability by offering homogeneous access to compute resources and unified virtual memory, the IVM framework shows performance comparable to the baseline even in the absence of load balancing.

When the number of PEs exceeds that of GPUs, the GPUs are oversubscribed. MPI-CUDA statically assigns the same amount of work to all GPUs. Charm-CUDA performs load balancing by migrating chares from overloaded to idle resources. MPI-CDYN and IVM performs load balancing by allowing PEs that terminate earlier to spawn more PEs and assign them work. As the number of PEs increases, the size of the work-portions decreases, leading to finer-grained load distributions and thus higher degrees of load balancing for MPI-CDYN, Charm-CUDA and IVM. However, excessively increasing the number of PEs leads to high migration overhead for Charm-CUDA and high DTC overhead for MPI-CDYN and IVM, resulting in performance degradation. In most of the cases where load balancing takes place, IVM performs better than the other implementations. The speedup of IVM over the other implementations increases with the number of PEs, and reaches an optimal point at 48, 36, and 32 PEs for NBODY, DMM and NW, respectively. The better performance of IVM compared to Charm-CUDA can be explained as follows. First, Charm-CUDA performs load balancing by migrating chares across nodes and GPUs. There are two sources of overhead associated with chare migration: rescheduling of the computation

Figure 5: Comparison between MPI-CUDA, Charm-CUDA and IVM (OM-LB). (a) Speedup over baseline MPI-CUDA with 10 processes. The number on the first bar indicates the execution time of the baseline MPI-CUDA code. (b) Relative load distribution across node types during the first iterations of HIMENO. Each bar represents the average load distribution for the nodes of any given type. The numbers above the bars indicate the standard deviation of the execution times of all PEs.

and data transfers between nodes. IVM minimizes the data transfers by allowing PEs to share data on compute nodes and it only initiates data transfers when data structure synchronization across PEs residing on different compute nodes is required. We found that, for our applications and datasets, the PEs can share 3-9% of the memory content across the compute nodes, resulting in reduced inter-node data transfers. Second, Charm-CUDA collects profiling information during the first few iterations of each application, and it uses the collected profiling information to perform load balancing in subsequent iterations. Therefore, some iterations in each Charm-CUDA application suffer from load imbalance. IVM, on the other hand, does not require profiling information to perform load balancing. IVM performs better than MPI-CDYN in most cases. This is because the MPI-CDYN versions of the applications do not allow PEs to reuse or share data structures, thus leading to communication overheads. However, MPI-CDYN performs slightly better than IVM in the case of NW with 32 PEs. This is because NW is a short-running application and is not communication-intensive. Finally, as discussed below, load balancing in MPI-CDYN and Charm-CUDA may not be optimal.

Effects on Load Distribution – Figure 4(b) shows the relative load distribution across different types of compute nodes for the IVM and Charm-CUDA experiments of Figure 4(a). Specifically, each bar represents the average load assigned to the nodes of a given type. For example, in case of Charm-CUDA with 32 chares, each of the two *type1* nodes is assigned about 10% of the overall load. We recall that the presence of heterogeneity among GPUs causes IVM applications using the DS-LB scheme to spawn PEs at different rates, thus introducing load balancing.

Based on characteristics (compute- vs. memory-bound) of the kernels within the considered applications, we expect the

performance of different types of nodes to be ordered (high-to-low) as follows: *type1-type3-type2* for NBODY and DMM, and *type2-type1-type3* for NW. MPI-CDYN, Charm-CUDA, and IVM can capture the performance heterogeneity and are able to distribute load to compute nodes according to their performance capabilities. However, we observe that in some cases Charm-CUDA fails to optimally assign the load to the nodes. In particular, in the case of NBODY with 64 PEs, Charm-CUDA assigns load to *type3* nodes equally to *type1* and *type2* nodes. In addition, in the case of DMM, Charm-CUDA always assigns less load to *type1* nodes (equipped with K40 GPUs) than to *type3* nodes (equipped with less powerful K20 GPUs). This inefficient load balancing is one of the limiting factors for the performance of Charm-CUDA applications in heterogeneous settings. In many cases especially in NBODY and DMM, MPI-CDYN distribute less work to *type1* nodes which can handle most of the work.

5.3 Online Monitoring Load Balancing

The IVM implementation of HIMENO uses the OM-LB scheme. Figure 5(a) shows the speedup of the MPI-CUDA, Charm-CUDA and IVM versions of HIMENO over a baseline MPI-CUDA implementation with 10 tasks (one per GPU). We set the size of the pressure grid to 2,048 elements for all directions (X-, Y- and Z-direction), and we set the threshold described in Section 4.1 to 1.5. In the case of MPI-CUDA, MPI-CDYN, and Charm-CUDA, we vary the number of PEs (x-axis). Recall that the OM-LB scheme monitors the execution of all PEs and periodically spawns more PEs on faster compute resources and redistributes the work across the PEs mapped to these resources. Therefore, in the case of IVM we do not directly control the number of spawned PEs; the numbers on the x-axis represent the maximum number of PEs that can be spawned over execution. As can be seen, IVM and Charm-CUDA perform similarly to the baseline for 10 PEs since in this case they do not perform load balancing. When the maximum number of PEs is 32 to 64, IVM outperforms MPI-CUDA, MPI-CDYN, and Charm-CUDA and achieves a 2.05x speedup. In all cases, IVM spawns a total of 25 PEs. OM-LB limits the DTC overhead by reducing the communication between IVM daemons for communication-intensive applications.

To understand these results, we monitor the relative load distribution performed by IVM across different types of nodes. Figure 5(b) shows how the load distribution varies in the first few iterations of HIMENO. The numbers above the bars indicate the standard deviation of the execution times of all PEs. We expect this number to be lower once the load becomes more balanced. In the first iteration the load is uniformly distributed across all types of nodes. In subsequent iterations, the application spawns more PEs on faster *type1* and *type3* nodes. More powerful GPUs are expected to be time-shared by more PEs. We confirm this intuition by measuring the standard deviation of the execution time of the PEs. As can be seen, this metric is high during the first few iterations of the application and decreases over its execution, as the OM-LB scheme increasingly submits PEs to the more powerful GPUs on *type1* and *type3* nodes.

5.4 Discussion

To complete our analysis, we discuss how the use of IVM dynamic task creation and one-sided communication mechanisms differs from the use of the same features provided in MPI-2.

Dynamic Task Creation – In IVM, all tasks (both statically or dynamically created) communicate through the shared memory abstraction. The Dynamic Process Creation mechanism offered by MPI has a more involved way to enable communication between

1	if (id == 0) { /* Root PE */	1	<i>MPI_Alloc_mem</i> (&var_in, ...); /* allocate shared input and output */
2	<i>ivmMalloc</i> (&var_in, ...); /* Input var. */	2	<i>MPI_Alloc_mem</i> (&var_out, ...); <i>MPI_Win</i> win[2];
3	<i>ivmMalloc</i> (&var_out, ...); /* Output var. */	3	if (id == 0) { /* Root PE */
4	} else { /* Working PEs */	4	<i>MPI_Win_create</i> (&var_in, ..., win[0]); /* create MPI windows for
5	<i>ivmMap</i> (&var_in, ...); /* Access input */	5	<i>MPI_Win_create</i> (&var_out, ..., win[1]); one-sides communication */
6	<i>ivmMap</i> (&var_out, ...); /* Access output */	6	} else { /* Working PEs */
7	<i>ivmSyncGet</i> (&var_in, ...); /* Read input */	7	<i>MPI_Win_create</i> (NULL, ..., win[0]); /* open window for var_in*/
8	... some computation ...	8	<i>MPI_Win_create</i> (NULL, ..., win[1]); /* open window for var_out*/
9	<i>ivmSyncPut</i> (&var_out, ...); /* Write output */	9	<i>MPI_Win_lock</i> (MPI_LOCK_SHARED, ..., win[0]); /* lock window */
	}	10	<i>MPI_Get</i> (var_in, ..., win[0]); /* read input */
		11	<i>MPI_Win_unlock</i> (0, ..., win[0]); /* unlock window */
		12	... some computation ...
		13	<i>MPI_Win_lock</i> (MPI_LOCK_EXCLUSIVE, ..., win[1]); /* lock window */
		14	<i>MPI_Put</i> (var_out, ..., win[1]); /* write output */
		15	<i>MPI_Win_unlock</i> (0, ..., win[2]); /* unlock window */ }

Listing 3: Pseudo-code for one-sided communication: IVM (to the left) and MPI (to the right)

dynamically spawned PEs, and this complicates the implementation of some load balancing mechanisms. In MPI, a parent-PE can spawn a set of child-PEs by invoking the *MPI_Comm_spawn* primitive. This primitive causes an inter-communicator to be created to enable communication between the parent and the child-PEs. However, if the child-PEs were to spawn grandchildren, it would be cumbersome to enable direct communication between the parent-PE and its grandchildren due to the lack of the required inter-communicator between them. This complicates the implementation of the DS-LB scheme depicted in Figure 3(a). To circumvent this problem, in our MPI implementation of DS-LB we added a set of master-PEs between the root-PE and the child-PEs. The master-PEs are initially created by the root-PE and distributed to different nodes, and their role is act as intermediaries between the root- and the child-PEs and request work-portions to the root-PE on behalf of the child-PEs. Once a child-PE terminates, the corresponding master-PE checks with the root-PE and spawns another child-PE until all work-portions are drawn from the root-PE. This implementation, however, adds communication overhead to the DS-LB scheme. Another difference between IVM and MPI is that, while in IVM each PE has a unique identification number, in the presence of dynamic process creation MPI ranks are not unique. As a consequence, MPI ranks cannot be used by PEs to identify work-portions, thus complicating the work distribution process.

One-sided communication – Listing 3 compares the use of one-sided communication in IVM and MPI. One-sided communication in MPI involves the use of epochs (defined by windows), which complicate the programming. Windows are created using the *MPI_Win_create* primitive (lines 5-6 and 7-8), and they must be explicitly locked before and unlocked after each *MPI_get* and *MPI_put* primitives. Not only does this mechanism lead to more complex code, it is also error-prone. In contrast, IVM offers more intuitive synchronized *put* and *get* primitives, leading to simpler and more compact code.

6. CONCLUSION

In this work, we have proposed Inter-node Virtual Memory (IVM): a parallel programming model for distributed GPU applications that offers a uniform view of compute resources and memory spaces. We have also described the design of a runtime framework that supports IVM. Our system, meant to increase programmability and reduce the complexity of application codes, offers Dynamic Task Creation (DTC) as a mechanism to facilitate encoding dynamic load balancing schemes within applications. We have designed two self-tuning load balancing schemes based on DTC and tested them on four applications. Our results, reported on a ten-node heterogeneous cluster, show that applications implemented using IVM can outperform both statically and dynamically scheduled MPI-CUDA and dynamically scheduled Charm-CUDA implementations.

ACKNOWLEDGMENTS

This work has been supported by NSF awards CNS-1216756 and CNS-1429294 and by a gift from NEC Laboratories America and equipment donations from Nvidia Corporation.

REFERENCES

- [1] Nvidia. GPU-Accelerated Applications Catalog. <http://www.nvidia.com/content/tesla/pdf/gpu-apps-catalog-mar14-digital-fnl-hr.pdf>
- [2] J. Kraus. An Introduction to CUDA-Aware MPI. <http://devblogs.nvidia.com/parallelforall/introduction-cuda-aware-mpi>
- [3] L. V. Kale and K. S., "Charm++: Parallel Programming with Message-Driven Objects," in *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds., ed: MIT Press, 1996, pp. 175 - 213.
- [4] V. Gupta, *et al.*, "GViM: GPU-accelerated virtual machines," Proc. of HPCVirt 2009.
- [5] G. Giunta, *et al.*, "A GPGPU transparent virtualization component for high performance computing clouds," Proc. of Euro-Par 2010.
- [6] J. Duato, *et al.*, "Enabling CUDA acceleration within virtual machines using rCUDA," Proc. of HiPC 2011.
- [7] V. T. Ravi, *et al.*, "Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework," Proc. of HPDC 2011.
- [8] M. Becchi, *et al.*, "A virtual memory based runtime to support multi-tenancy in clusters with GPUs," Proc. of HPDC 2012.
- [9] K. Sajjapongse, *et al.*, "A Flexible Scheduling Framework for Heterogeneous CPU-GPU Clusters," Proc. of HiPC 2014.
- [10] D. Sengupta, *et al.*, "Scheduling multi-tenant cloud workloads on accelerator-based systems," Proc. of SC 2014.
- [11] B. Acun, *et al.*, "Parallel programming with migratable objects: charm++ in practice," Proc. of SC 2014.
- [12] Unified Parallel C. <https://upc-lang.org>
- [13] R. Brightwell *et al.*, "An intra-node implementation of openshmem using virtual address space mapping," Proc. of PGAS 2011.
- [14] M. Bauer, *et al.*, "Legion: expressing locality and independence with logical regions," Proc. of SC 2012.
- [15] H. Kaiser, *et al.*, "HPX: A Task Based Programming Model in a Global Address Space," Proc. of PGAS 2014.
- [16] The Chapel Parallel Programming Language. <http://chapel.cray.com>
- [17] X10: Performance and Productivity at Scale. <http://x10-lang.org>
- [18] Unified Memory in CUDA 6. <http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6>
- [19] OpenCL 2.0 - Shared Virtual Memory. <http://developer.amd.com/community/blog/2014/10/24/opencl-2-shared-virtual-memory>
- [20] J. Dinan, *et al.*, "Dynamic Load Balancing of Unbalanced Computations Using Message Passing," Proc. of IPDPS 2007.
- [21] D. Li, *et al.*, "A Distributed CPU-GPU Framework for Pairwise Alignments on Large-Scale Sequence Datasets," Proc. of ASAP 2013.
- [22] S. Olivier, *et al.*, "UTS: an unbalanced tree search benchmark," Proc. of LCPC 2007.
- [23] A. M. Sidelnik, *et al.*, "Performance Portability with the Chapel Language," Proc. of IPDPS 2012.
- [24] S. Potluri, *et al.*, "Extending OpenSHMEM for GPU Computing," in IPDPS 2013.