Distributed Computing Environment 3.0 Functional Overview

White Paper

Document version 3.0.1
29 May 2012

Document build number 170068

Table of Contents

1 Overview 1
2 Our Goals 2
  2.1 Solutions 2
3 DiCE Feature Overview 2
  3.1 High Level API 2
  3.2 Interactive Applications 3
  3.3 Multi-core and Clustered Operations 3
  3.4 Multi-user Operation 3
  3.5 Data Sharing 4
  3.6 Distributed System 4
  3.7 Redundancy 4
  3.8 High Performance Networking System 5
  3.9 Self-Organizing Clustering 5
  3.10 Database Elements 5
  3.11 Distributed Reference Counting 6
  3.12 Transactions 7
  3.13 Job System 8
  3.14 API 9

1 Overview

NVIDIA's Distributed Computing Environment (NVIDIA DiCE) is a platform for writing parallelized high performance computing applications. It helps developers to write applications that scale to thousands of multi-core CPUs, possibly combined with thousands of GPUs. DiCE addresses both the scaling to the CPUs and GPUs of one machine as well as leveraging all the resources in a cluster of machines.

DiCE was written in a way that enables application developers, who are not experts in multi-threaded programming or distributed systems, to obtain highly scalable and robust applications while allowing them to concentrate on their field of expertise.

Compared with other existing solutions, intended to simplify the development of scalable high performance computing applications, DiCE has many unique aspects that makes it suitable to a much broader field of problems:

- **High level API**: DiCE has a higher level API in comparison with systems like MPI (Message Passing Interface). The system does not require application developers to deal with low level communication aspects but allows them to store data and submit computing tasks, which are then automatically executed in a cluster.
- **Interactive applications**: With DiCE, you can implement applications that perform massive numbers of calculations that need to be finished after only a few tens of milliseconds. Indeed, it is possible to distribute the workload to a large cluster of machines.
- **Distributed database**: DiCE has a built-in distributed database that transparently handles the data distribution to all nodes in a cluster and provides many other features, which help to write applications operating on that data, such as fault tolerance.
- **Automatic load balancing**: Built-in scheduling algorithms automatically take care of the load balancing of various resources within a cluster. This enables these resources to be used efficiently. In cases where domain-specific knowledge is available, the application overrides the built-in scheduler with tailored scheduling decisions.
- **Support for multi-user operation**: DiCE supports the concurrent running of multiple operations, thereby allowing the mixing of long- and short-running operations on the same cluster. With DiCE, concurrent tasks can share all or parts of the data needed for their computations, thus drastically reducing the memory consumption of the system.
- **Distributed system**: The DiCE architecture is by far superior to simple client/server systems: DiCE is a truly distributed system. Owing to this, it is possible to implement a much wider range of application topologies, which are suited to more usage scenarios. Furthermore, this architecture enables DiCE to provide failure safety: the handling of machine failures without any interruption to service.

2 Our Goals

In the past, and in accordance with Moore's Law, computer CPUs were getting increasingly faster every year. While this is still a valid argument for the computing power of one CPU, nowadays, increase in speed is mostly achieved by adding more cores to a single CPU. Therefore, compute-intensive applications that do not make optimal use of multiple cores, no longer reap the full benefits of the advancements made in the speed of CPUs.

However, even if an application is able to successfully scale to many cores on one machine, in many cases this is not sufficient for compute-intensive tasks that would require hundreds or even thousands of cores. As a consequence, leveraging the computing power of a networked cluster of machines is getting increasingly important in the area of high-performance computing.

With the advent of programming environments such as CUDA^{TM}, GPUs can now also be utilized to speed up computing applications. Again, although it is possible to equip machines with several GPUs, it is the application programmer's responsibility to ensure that they are all efficiently used. Using GPUs in a networked cluster can help to scale the computation to many more GPUs than is possible with a single machine.

2.1 Solutions

DiCE originates from the observation that specialists in a certain area of software development have the domain-specific knowledge required to write optimized code in their domain. Likewise, the latter usually tend to know how to parallelize their code from an algorithmic point of view. Nevertheless, in many cases these specialists are not experts at implementing their algorithms in a way that would efficiently leverage the many cores of modern machines. Few application developers know how to write software that is able to run on many machines in a cluster as a single, massively parallel system.

Indeed, writing a distributed system that is scalable to hundreds of hosts with thousands of multi-core CPUs, and robust enough to be used in production systems, is a very complex task. It is something that, even for specialists in distributed computing, may take years to realize. In addition, the ongoing maintenance of such a software is vastly complex, resource-intensive, and may require, for example, the support of new operating system and driver releases. DiCE provides the infrastructure to relieve application developers of this burden.

NVIDIA's Distributed Computing Environment comes as a library that can be linked to applications and offers an easy to use API to distribute work to all CPU cores and GPUs in a huge cluster of machines. As a result, there is no need to worry about threading, load balancing, network operations and so forth. DiCE simplifies the scaling aspect of high-performance computing applications for developers, thus enabling them to concentrate on the core aspects of their work and their field of expertise.

Although there are several systems available today that are meant to help application developers implement scalable, high-performance computing applications, DiCE has many unparalleled features that make it uniquely suitable to a much broader field of problems. These features are explained in more detail in the following sections.

3 DiCE Feature Overview

3.1 High Level API

In contrast to low level systems like MPI (Message Passing Interface), DiCE has a higher level API that does not force application developers to worry about things like message passing and low level communication. Instead, with DiCE you can store objects in a distributed database and formulate computing tasks that use those stored objects to do calculations. DiCE handles all synchronization, scheduling, workload balancing and other related aspects for the application.

For example, an application can store data in the DiCE database and then submit a task that needs this data. When parts of the computation are done on other nodes in the same cluster, the application writer does not have to care about the availability of that data on the executing node. The DiCE system guarantees that all of this happens before or when the data is accessed. This is true even if changes are applied to database elements: DiCE will make sure that the correct version of the database element is accessed on each node.

3.2 Interactive Applications

Many systems available to do clustered computing are suitable for applications where a single computation takes several minutes or longer. DiCE originated from interactive 3D visualization technology. Due to this, it was specifically designed to allow the clustered computation of tasks that should be finished after a few tens of milliseconds.

This is not too difficult to achieve with small numbers of computing resources. It is also relatively easy to achieve for applications where there exists a known method to automatically split workloads into small parts that have roughly the same run time. However, it is very difficult to implement this if the mentioned prerequisites are not met.

DiCE is optimized for splitting tasks into thousands of parts of possibly diverging execution time. This is the basis for scaling to very many cores in a cluster.

3.3 Multi-core and Clustered Operations

There are various middleware systems that simplify the usage of multiple cores in one machine. Most of them do not take into account how to extend this to a whole cluster of machines. Thus, porting an application that already scales well on a multi-core machine to a cluster results in a substantial rework of the application if not a complete rewrite of it.

The design of the DiCE API enables you to implement software that runs on a system with multiple hosts almost as easily as on a system with multiple multi-core CPUs. Basically, by extending a certain algorithm that uses DiCE to run on multiple CPU cores of one machine to run in addition on multiple nodes in a cluster is in many cases a simple extension to the implementation. The fundamental principle is the same.

In addition, the support for clustered operations does not constitute overhead when the software is used on a single machine. DiCE has special code paths that are taken in case an operation is executed locally.

3.4 Multi-user Operation

Current clustered computing systems are designed with a queuing system to implement access to multiple users. Such systems have one central entry queue where job submissions are entered using the API of the system. A submitted task waits until it is scheduled to run on the clustered system because the system can only operate on one task at a time. Considering that some tasks may take minutes or even hours to complete, the waiting time for each task may be very high even if the task itself finishes in a short amount of time. This is the case even if the system is able to prioritize tasks.

With DiCE in contrast, unlimited operations can be run in parallel. This means that a short running task submitted after a long running task is able to overtake the long running task.

With multi-user support, you can use all available resources in a cluster more efficiently. In spite of efforts made to parallelize computing tasks for many algorithms, there is often a fan-out phase in the computation where gradually fewer cores have work left to do. Other cores become idle at that time. With multiple operations overlapping each other, the system automatically uses idle resources for other tasks in such a case. Thus, DiCE can provide a much higher overall throughput and resource usage level.

3.5 Data Sharing

Another aspect of multi-user operation is the sharing of data. In many cases a certain, possibly extremely large dataset, is used by multiple members of a team at the same time to perform similar calculations. Imagine a large city model that is used to simulate traffic flow. Different team members could perform simulations based on the same city model but simulate for different times of the day or different days of the week.

Instead of loading the dataset individually for each team member, DiCE allows them to share the same dataset for their tasks. Of course it is possible that only parts of the dataset are actually shared by a larger working set. This allows for great savings of main memory while still giving flexibility to the creation of datasets from smaller parts.

DiCE also supports the concept of multiple versions of the same data. This means that it is possible to change small parts of a huge dataset and obtain different variants of them for each user. The distributed database built into DiCE automatically blends in the correct version of each database element for each user when the calculation accesses it. Therefore, instead of having to replicate a huge dataset to change only a small part of it, you can share most of the memory while still having small individual deviations. In the aforementioned traffic simulation scenario, individual simulations running at the same time could apply changes to certain parts of the city and run simulations based on these changes. This would provide information to find out what impact they would have on the traffic.

3.6 Distributed System

DiCE is not based on a simple client/server architecture. The DiCE architecture is a genuine distributed system where each node in a cluster is conceptually equal to all other hosts in a cluster. This means that DiCE does not have a single entry point into the system that is responsible for all user interaction and for aggregating all results, thus causing a potential bottleneck and a potential single point of failure. Instead, each node in a cluster can be an entry point to an application written with DiCE.

This is important because for many computing tasks the head node that accepts the request to execute a task also has a considerably higher workload to collect results from the other hosts and aggregate them. This applies to CPU, memory and network resources. With DiCE, incoming requests can be distributed to all the hosts in the cluster. Therefore, the system utilizes the available resources to a much better extent.

Note that although each node in the cluster is conceptually equal to all others, they do not necessarily need to have the same roles. For example, you can easily have a number of workstation nodes that are used by users to access the system, and other background server nodes that are simply used to speed up computations. DiCE is highly configurable so as to accommodate many different usage scenarios. For example, you can configure DiCE so that the workstation nodes do not accept work from other nodes, and therefore they do not block their resources for other work by the user. This can be changed dynamically to accommodate changes. For example, if a user logs out, you can instruct DiCE to make full use of the available resources.

3.7 Redundancy

In clusters of nodes built from commodity hardware, the probability that at least one of the nodes fails because of hardware problems, rises with the size of the cluster. For big clusters, this probability might be too high to be tolerable for certain applications. To address this problem, DiCE provides a failure detection and recovery system based on redundancy. Since the distributed database in DiCE has full control over the distribution of data across the system, it can guarantee that at any point enough nodes in the cluster have a copy of a certain database element. Therefore even the simultaneous failure of one or more nodes can be overcome without data loss and with hardly any perceivable impact on the user's experience.

The DiCE database will also ensure that the required redundancy is reached again as soon as possible after the failure of nodes. For this, some of the nodes will acquire copies of database elements that do not have enough copies left in the system to be able to further guarantee the required redundancy. This is called a recovery period.

During a recovery period, operation proceeds without interruption but possibly with slightly reduced performance. Once the recovery period has finished, operation returns to normal and the failure of further nodes can be tolerated again without problems.

You can add new nodes into a running system to replace failed nodes, or further enhance the processing power of the system. The DiCE system will make sure that new nodes gather all necessary data needed to execute work that has been assigned to them. This happens without any knowledge or interference of the computing application.

3.8 High Performance Networking System

DiCE implements a high performance networking layer with several modes of operation. The default mode is based on the UDP protocol and leverages multicast transmissions to provide for high efficiency. Multicasting lets a certain node send data only once although a whole set of nodes will receive the data. When this happens, the network infrastructure replicates data and this mode optimizes the usage of network bandwidth while minimizing the usage of CPU needed for network operation. Thus UDP multicasting is the mode of choice because it allows to scale to the highest number of nodes.

DiCE provides more modes of operation based on UDP but without multicasting. It also provides a mode based on the TCP/IP protocol. This enables you to use DiCE in network environments where the network infrastructure does not support the more advanced modes. This has no impact on the application based on DiCE except for possible scalability differences of big clusters.

All network modes can operate on Ethernet networks with any speed as well as on Infiniband networks. In general, every network that allows the operation of the standard Internet protocol is suitable for the deployment of DiCE.

3.9 Self-Organizing Clustering

DiCE can provide for configuration-less operation using a multicast based node discovery protocol. The application integrating DiCE does not have to provide a list of nodes that should be connected together.

Instead, each node simply needs to know a multicast address for the group of nodes it should collaborate with. The node will then use multicast transmissions to automatically find all the other nodes in the cluster. The DiCE network uses self-organization based on sophisticated protocols to get a common understanding of required information such as the list of all nodes in a cluster needed to guarantee the consistency of the database.

Self-organizing clustering allows nodes leave a cluster to do maintenance and re-join the cluster later. It also allows to dynamically adapt the size of a cluster to the load on the cluster.

3.10 Database Elements

The database stores C++ objects. Every kind of C++ object can be stored in the database if the author of the C++ class provides a few easy to write member functions such as: serialization, deserialization, the duplication of an existing instance of that class and the creation of an empty instance.

Each stored C++ object results in what is referred to as a database element. Database elements are addressed using numeric identifiers and string names.

The DiCE system automatically distributes database elements to other nodes in the network. This is done proactively, but it can also be performed on demand. Each node caches a subset of all versions of all database elements at the same time. The caching allows for very fast operation.

The database can also evict data from the local caches of a node to be able to cope with memory shortages. Based on the knowledge that a sufficient number of other nodes has a certain database element, a node can drop database elements it has not used for a period of time to make room for needed data. This is done automatically and does not require any interaction with the user application or the user.

Below is an example of a simple database element that stores an STL string in the database. The class needs to be registered with the database. Subsequently, instances of this class can be stored in the database. The database can transport them to other hosts in the cluster without further requirements on the application. The application can then retrieve the data from the database and operate on it.

```
class My_db_element : public mi::neuraylib::Element<...>
{
public:
    void serialize(mi::ISerializer* serializer) const
    {
        serializer->write(&m_a, 1);
    }

    void deserialize(mi::IDeserializer* deserializer)
    {
        deserializer->read(&m_a, 1);
    }

    mi::neuraylib::IElement* copy() const
    {
        return new My_db_element(*this);
    }

    std::string m_a;
};
```

The class needs to be registered with the database:

```
dice_configuration->register_serializable_class<My_db_element>();
```

3.11 Distributed Reference Counting

With DiCE, database elements can contain references to other database elements. These references can be expressed using the numeric identifier of a database element. These numeric identifiers (named "tags") act as pointers in more conventional systems to express references between database elements.

Given that DiCE permits multiple operations to run at the same time and multiple versions of database elements to exist at the same time, special attention needs to be paid to the lifetime management of data when taking these references into account.

In general, it is very difficult to decide for a distributed system with parallel operations originating on different hosts which data has become obsolete and can be removed when each operation can make changes to stored database elements. Simple approaches that work on single node or client server systems do not work in a distributed system.

The DiCE system thus implements a sophisticated distributed reference counting mechanism to guarantee the correct lifetime of database elements. The DiCE system handles this automatically only based on the list of tags referenced by a database element. This list needs to be provided by a member function of the database element. DiCE makes sure that any unreferenced and thus obsolete database elements are removed from the system.

Below is an example of a simple database element that stores a tag. The tag is a reference to a different database element. The other database element will automatically be kept alive as long as the referencing instance is still in the database. The only change to the previous example is the additional `get_references` function.

```
class My_db_element : public mi::neuraylib::Element<...>
{
public:
    void serialize(mi::ISerializer* serializer) const
    {
        serializer->write(&m_a, 1);
    }

    void deserialize(mi::IDeserializer* deserializer)
    {
        deserializer->read(&m_a, 1);
    }

    mi::neuraylib::IElement* copy() const
    {
        return new My_db_element(*this);
    }

    void get_references(mi::neuraylib::ITag_set* result) const
    {
        result->add_tag(m_a);
    }

    mi::Tag m_a;
};
```

3.12 Transactions

The database provides support for transactions with Atomicity and Isolation properties. Atomicity means that a transaction can be committed or aborted at any time. If a transaction is committed, all changes to the transaction become visible at the same time for all nodes. If the transaction is aborted, all changes in the transaction are automatically abandoned and will not be visible for any other transactions. Isolation allows concurrent transactions that do not influence each other at all.

Transactions are a fundamental part of the DiCE system. They provide the foundation upon which the multi-user aspects of DiCE have been built. Indeed, transactions fully work on clustered systems, and each of the nodes in a system is able to start and commit transactions and use them for operations.

Note that DiCE does not provide the Durability property of ACID databases. All storage is done to main memory for fast operation. In the case of memory overusage, DiCE can offload stored data to a disk. This is not meant to be done for persistent storage. Using external code, you can write data to conventional databases or plain files or read data from them .

3.13 Job System

The core purpose of the DiCE infrastructure is to enable the efficient execution of computing work on all available compute resources in a cluster. The job system in DiCE is the basis for this. This system formulates work by implementing a C++ class containing execution functions that are called to do the computation work. Instances of these classes are passed to the job system and the execution function of these instances is called whenever and wherever the DiCE scheduling system considers this to be necessary. Jobs come in two basic flavors:

- Database Jobs

  These jobs are stored in the database. They can be accessed by the application just like any database element. Basically, database jobs are computational database elements. The content of the database element is created on demand.

  A database job can be accessed by the application code many times on all nodes and cores in a cluster during the processing of one transaction. However, the DiCE system will make sure that the job is executed at the most once and only if actually needed. Therefore, if a certain job is not accessed at all during the computation in one transaction, it is not executed at all (lazy execution).

  Database jobs can also be marked as shared jobs. Once they have been executed in one transaction they will be shared by all other transactions that access the same job. It is only when the application actively invalidates the results of a job that it will be re-executed in future transactions.

  The transmission of the job data to other nodes in the cluster and the transmission of any job results back to the originating node is automatically done by the DiCE system. This transmission is based on a small set of support functions written by the application developer similar to those needed for database elements. All necessary synchronization is done in the background by the DiCE infrastructure. This ensures that the required data is available on the executing node and that the correct version of the data is used.

- Fragmented Jobs

  A fragmented job is not stored in the database. It may be created on the fly and then given to the DiCE system for immediate execution. In contrast to database jobs, one fragmented job represents a work task which can be split into many work fragments. At submission time, the application can specify how many fragments the job should be split into. The application needs to implement a C++ class with execution member functions. These functions will be called once for each of the fragments on one appropriate computing resource in the cluster. The DiCE system provides transport and synchronization.

  The fragmented job system is optimized to execute fragmented jobs composed of several thousand fragments in a fraction of a second on a big cluster of machines. This forms the foundation for the scalability of applications based on the DiCE system.

  Below is an example of a simple fragmented job that calculates functions for various input values on a cluster. The `execute_fragment` function is called for each fragment on the local host. The `execute_fragment_remote` function and `receive_remote_result` function form a pair that should have the same effect as the `execute_fragment` function on the local host: `execute_fragment_remote` is called on a different host in the cluster and calculates a result that is then transmitted to the requesting host. Then DiCE calls the `receive_remote_result` function which integrates the result into the overall result, which is similar to what the `execute_fragment` function does.

  All kinds of complex data including C++ classes can be transported to the remote hosts and back in the result. The fragmented jobs can access the database to retrieve their respective input data.

Fragmented jobs can also invoke other fragmented jobs thus resulting in the hierarchical partitioning of data or work.

```
class My_fragmented_job : public mi::neuraylib::Fragmented_job<...>
{
public:
    static const int test_size = 1024;

    void execute_fragment(
        mi::neuraylib::IDice_transaction* itransaction,
        mi::Size index,
        mi::Size count)
    {
        m_results[index] = index * count;
    }

    void execute_fragment_remote(
        mi::ISerializer* serializer,
        mi::neuraylib::IDice_transaction* transaction,
        mi::Size index,
        mi::Size count)
    {
        Uint64 result = index * count;
        serializer->write(&result, 1);
    }

    void receive_remote_result(
        mi::IDeserializer* deserializer,
        mi::neuraylib::IDice_transaction* transaction,
        mi::Size index,
        mi::Size count)
    {
        deserializer->read(&m_results[index], 1);
    }

    Uint64 m_results[test_size];
};
```

While DiCE does not provide specific support for CUDA-based GPUs, the design of the DiCE system enables you to use CUDA to implement computations in a DiCE-based application. DiCE ensures that all the GPUs in each node in a cluster are used as efficiently as possible. Similar to the situation with CPUs, DiCE distributes work to all nodes in a cluster that have installed GPUs.

DiCE-based applications use CUDA-enabled GPUs to offload calculations to the GPU. The scaling to many GPUs works very well in these cases.

3.14 API

DiCE is accessed using a modern object-oriented C++ API. DiCE is designed to be portable to many platforms. Currently it is available on Windows, Linux and Mac OS X in 32-bit and 64-bit versions. Other operating systems can be supported easily.

DiCE is supplied as a shared library and as a library that can be statically linked to the application. The API has been carefully designed to allow safe operation of memory allocations and deallocations even in the shared library case. The API also takes into account later extensions with backwards compatibility.