Leopold–Franzens–University
Austria, Innsbruck
&
Japan Advanced Institute
of Science and Technology
Japan, Ishikawa

Institute of Computer Science
Research Group:
DPS@LFU
(Distributed and Parallel Systems)
Research Group:
Inoguchi Lab@JAIST
(Security and Network Area)

Distributed GPGPU on Cloud GPU Clusters

Master Thesis

Supervisor: Dr. Juan J. DURILLO
Prof. Dr. Yasushi INOGUCHI
Dr. Sashko RISTOV

Martin Schuchardt

martin.schuchardt@student.uibk.ac.at

Innsbruck
1 July 2018

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Unterschrift

Acknowledgements

At this point, I would like to thank my three supervisors for their support and patience. Since I wrote this thesis as part of a term abroad, it was necessary to start and finish it at the LFU in Innsbruck and working on a predefined, separate part at JAIST in Japan. Compared with other theses, this added a lot of additional administrative tasks, made communications more difficult and prolonged the processing time.
My supervisors knew that, accepted me nevertheless, and helped me realize my term abroad, expand my horizon, and work on a cool, future-oriented subject with a multicultural team.

Thank you very much, for your support, your knowledge, for great ideas and feedback. Muchas gracias, どうもありがとうございました, ти благодарам многу, Dankeschön!

Abstract

Using GPGPU for problems, which can be solved via massive parallel algorithms, may lead to performance gains on appropriate hardware.
The high number of instances rentable from cloud providers like Amazon EC2[1] offer an interesting basis for powerful distributed systems. Combining both technologies could provide an immense computation power if the problem scales well.
The goal of this Master thesis was writing a library that simplifies and automates the creation of a cluster as well as typical requirements for distributed systems like send/receive, broadcast, execution, ...
To prove the usability of the library, real applications that are distributable and already use GPGPU should be implemented and benchmarked.

[1]https://aws.amazon.com/ec2/instance-types/

Contents

List of Figures

List of Tables

1 Introduction

The core idea for this thesis is to develop a library, that solves a problem, described as a workflow (a graph), in parallel, on multiple servers with GPUs. This graph may not contain cycles, and each node covers some consecutive parts of the code, required to solve the problem, as shown in figure 1.1.
It is unavoidable to have some overhead if parallelization is added to an existing algorithm. The library that has been developed for this thesis should take some of this overhead, but provide some useful features with as minimal as possible overhead.
More details can be found in chapter 2, but for a brief description the following section 1.2 will be helpful.

1.1 Motivation

Many real life problems in computer science are designed to be represented as a workflow. Our framework follows that idea, a graph with no cycles (which is a DAG) is the classical way of representing workflows.
Typical HPC installations provide a single scheduler which is responsible for assigning compute nodes. The scheduler requires to have all hosts preinstalled and configured. An executed program can specify what type and how many resources it requires, but has no influence on when or where it will be executed. The scheduler with all its prerequisites is mandatory.
We save on this prerequisites and generate the cluster from scratch. The executed program does not notice much difference, but the host, which has started the program and is using our library, is in full control of the cluster.

Our library allows execution on the Cloud and GPUs. This is especially relevant nowadays as a big percentage of applications which run on a supercomputer are AI based (and more specifically deep learning ones). Deep learning can be accelerated by GPUs because basically, many of the operations to be done in deep learning algorithms, such as convolutions, can be reduced to one or several chained matrix multiplications. We have used our library in the usability-tests by distributing matrix multiplications on GPU clusters - with a noticeable reduction of effort and complexity for the developer, as shown in chapter 5 and 6.

Potential users of our work have interests in the field of physical modeling, cryptography, and cryptanalysis, AI, deep learning, computations on a non-trivial amount of data, or pattern recognition, just to name some important ones. Not all of them have the money necessary to host a GPU cluster on-site or enough workload to keep a cluster 24/7 busy. Powerful clusters are expensive, and idle times waste money. Using the Cloud may either be more economic or make high-performance computation for short-term requirements possible within an acceptable budget.

1.2 Nomenclature

In the remaining documentation, several names will be used. We introduce all of them in the following list. They will be written in italic font from their first mentioning until their definition.

- A *node* defines a logical grouping of code within the whole program. For example initializing the matrices is a part of the program, and grouped in the node "init input matrices" (see figure 1.1).

- Each node has two *kernels*: one will be executed by the *master*, one by *workers*. We believe that many programs will have significant differences in the code that the master and the workers execute, splitting the kernels mimics this. Kernels are function pointers with this signature: $int(Node\&, Distributor\&)$.
  A node can have different kernels for *master* and *workers*, but also use the same or have a null pointer as kernel. In the latter case, nothing needs to be computed for that node. For example, the verify-node has nothing to do on *workers*, only the *master* has to compute some code. An example where the kernels are completely different are described in chapter 5, the compute kernels for the matrix multiplication: the *master* distributes chunks and collects results, the *workers* compute chunks on the GPU. These two functions have almost nothing in common.

- Regarding *Data* (arguments): since the interface for a kernel is fixed, additional arguments that might be needed are bound to the corresponding node or to the *distributor* (helpful if you need access to the same argument in various nodes). You can add as many arguments as you need, also with different types. They are stored as a map of *data*-objects, with the following signature: $Data(void*, vector < unsigned >, unsigned)$.
  The first argument points to the actual data, but since we want to store any kind of type and put all possible Data-objects together in containers of arguments, we need to replace the type by void*. The second specifies the number of elements. We use a vector in case the programmer wants to store dimensions, like for an NxM matrix. The third specifies the size of one element, which is required since we need to use void pointers and several MPI functions need to know the number of bytes to be transfered.
  Using templates instead of void pointers would not have been helpful because we wanted to have identical types, so that we can store arguments of different types in the same map, for example one argument that points to the dimension of a matrix ($unsigned\, N$), and another that points to the actual data ($double\, matrix[N * N]$).

- The *distributor* is a class that controls start/restart/termination of the cluster, solves the graph by executing the nodes, manages output, internal data like MPI rank and size, and much more [3].
  Because it is the most important class in the library, we will use "Distributor" as a synonym for the library.

- One process on one host starts and controls the program. This process is called *master*. Its tasks are parsing the *mpi.hostfile* of an existing cluster for names and number of *workers* or launching a new cluster with the requested number of *workers*. When it comes to distribution, the this master process uses the library and maintains a queue of idle *workers*, to which the master can distribute chunks of work. If all *workers* are busy, the master needs to wait for finished *workers*, collects computed results, and then may reuse the now idle *worker* for additional computations.
  The library supports shrinking of the cluster. This is also under the control of the master process. After all computations have finished, the master sends a termination signal and collects final logging data from *workers*. Optionally, unused *workers* will be shut down by the master.
  Nodes, that have a kernel for the master process only, are drawn in blue color in the computation graph as shown in figure 1.1.

- One or more *workers* are required for computations. By default, the first worker on an *instance* uses GPU 0 for computations, the second worker GPU 1, ...
  Workers are not designed to keep any data after the computation has finished and may be shut down and destroyed as soon as the master decides that the worker will not be needed anymore.

- One physical or virtual server can host multiple workers and one server needs to (additionally) host the master process. We will continue with the name *instance* for a server that hosts processes.

- The *mpi.hostfile* is a text file that contains the hostnames (or IP addresses or hostnames) plus the number of MPI-processes that will be started on that instance. Here, one MPI-process maps to one worker. The master is an MPI-process too, but already running, and right now using the library to initiate the start of the workers. Treat (MPI-)process as what physically happens, master and worker as logical definitions for the concept of *distribute&control* and *compute only*.
  The name of the file has to be *mpi.hostfile* and it needs to be in the same directory as the main executable. It might make sense to define the number of processes per instance equal to the number of GPU devices on that instance (parameter: *slots=<int>*). However, overloading is supported.
  But the cluster needs to be homogeneous, and the number of GPU devices (defined as *slots* in the *mpi.hostfile*) needs to be identical for all instances. We want MPI to add the new MPI-processes by filling up node by node. Later, we only know the ranks of the processes, not the instance on which they are running, but

we need the mapping of instance/corresponding MPI-ranks when we resize the cluster. Therefore, our library needs this constraint of having an identical number of processes on all instances.
Except for the instance that hosts the master process. This machine may host the master process additional to worker processes (→slots +1) or the master process only (→slots=1).

- A node will have *dependencies*. For example, "verify" cannot be executed before "computing chunks" has been finished. These dependencies are displayed as arrows in the computation graph in figure 1.1.
  The library computes a problem by starting at the target - in this case "verify" - and tries to solve all dependencies recursively. Nodes without dependencies like "init input matrices" can be computed immediately.

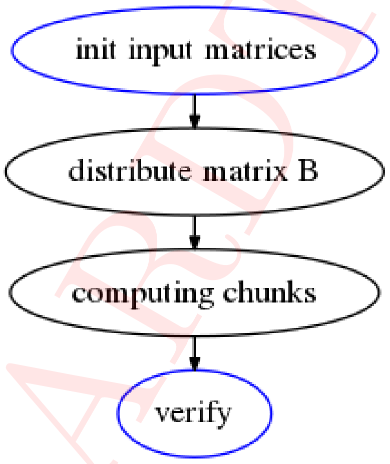


Figure 1.1: Graph for a simple matrix multiplication

For more explanations see chapter 5, which discusses the implementation using a naïve matrix multiplication and many of the features the library offers.

1.3 Cluster

Each cluster needs to host the master and at least one worker node, and so needs to have at least one instance. On the instance that hosts the master process, probably some additional services and tools might be required like compilers, shared directories for remote access and tools to create the cluster like the *aws command line interface*. Such a special instance can be specified and the required tools will be automatically installed and configured.
The main target for cluster instances are Linux servers with GPU devices on Amazon EC2. But for development purposes, cluster instances can also be launched on virtual

machines, hosted by VirtualBox[1] on a developers workstation. Hardware requirements are low, even underpowered, older devices can easily host several cluster instances and allow development locally.
One of the main goals of this thesis was to automate the creation, configuration and termination of a cluster. However, it is also possible to reuse existing cluster instances like a locally installed virtual box cluster. Amazon AWS instances are paid per hour, so for saving money it is supported to start and pay for instances only for the time the computations run and terminate them as soon as possible.
Both AWS and VirtualBox need some prerequisites. They are described in chapter 3 and 4. A script *setup/setupDistributedGPGPU.sh* is provided to help setting up a test cluster and the prerequisites for the library. Please do not think of the install script as a one-click installer. It is separated into 10 small blocks so that if one fails, you can fix whatever difference in your environment made the script fail, restart the script and continue after skipping all already completed steps. There are some basic checks that subsequent calls of the script skip already completed actions, but in general, this setup-script has been designed to be an installation guide.

1.4 Implementations

To test the usability of the library, two algorithms have been implemented. The focus was not on achieving a better performance by using single instances and fewer GPU devices, we only wanted to demonstrate how to rewrite existing programs to make use of our library and so of clusters. We aim to provide a tool that helps to execute code on cloud clusters, reduce time spent on necessary prerequisites or maintenance of the instances before the actual computation can happen, and then helps to achieve a better performance by using multiple GPUs in parallel.
Chapter 5 describes the library functions and features on a simple, naïve matrix multiplication code sample. Chapter 6 shows the implementation of a real world problem, a sparse matrix-vector multiplication (SpMV) using the generalized minimal residual method (GMRES), that kindly has been provided by JAIST and Fujitsu.

[1] www.virtualbox.org

2 The Library

The main idea for the library was simplification. Adding parallelization, for example by using MPI, always adds additional code and complexity. We tried to hide general parts of the code in the library, like for starting/stopping processes, wrapped jointly used parameters and functions, for example, IDs and communicators for data transfer, or added handy features like quieting console output and instead of disordered, parallel writes on the console from several processes, we collect all processes output and print them in order at the end of the program.
In the backend, MPI is used. All functionality that OpenMPI offers is usable by the programmer. Hence, the library will not restrict any possibilities, only add some more comfort. At least, this is the idea: a proper prepared and suitable program should be able to be rewritten to be parallelized and be executed on an EC2 GPU cluster with minimum effort.

The heart of the library is the Distributor-class. It handles the start/stop of processes, computes the graph that describes the problem, offers states like MPI rank, handles output (silent mode), cluster resize and much more. This class has the biggest complexity and hosts most functions and parameters that a programmer might want to use.
Closely linked is the Node-class. Nodes encapsulate different parts of the problem, like initialization, distribution, and computation of your data. The graph, the Distributor works on, is described with these Nodes. So, while initializing the cluster with help of the Distributor, the programmer hands over the Node-objects to the Distributor to solve the problem.
Next, for simplification, all data that the Nodes need to exchange with other instances in the cluster, are wrapped into objects of the Data-class. This class offers various MPI data transfer methods, and they have been wrapped into meaningful functions and reduce the number of required arguments to a minimum so that the programmer does not need to take care of sender id, communicators or received number of objects.

The next sections will describe the most important member functions of these classes. This chapter has not been written to be a complete documentation of the library, but to help understand the basic idea. The code snipped showed in listing 2.1 should help understanding how the library works by using the most important functions of the library. When starting the program that will use the library, we recommend using something similar to the tiny script we provide with the Makefile for our sample matrix multiplication: *make run* or *make runSilent*. Because of some MPI specifics, calling the program is a bit tricky as described in section 2.1.4 and section 2.5.

```
int kernel_DistributeB(Node &n, Distributor &d) {
  auto err = d.getArgument("B")->bcast_M_to_W();
  return err;
}

int kernel_Verify(Node &n, Distributor &d) {
  const unsigned N = *static_cast<unsigned*>(d.getArgument("N")->get());
  int *C = static_cast<int*>(d.getArgument("C")->get());

  auto err = testResults(C, N);
  return err;
}

int main(int argc, char** argv) {
  Node *distributeB = new Node("distribute matrix B", kernel_DistributeB,
    kernel_DistributeB);
  Node *compute = new Node("computing chunks", kernel_ComputeOnMaster,
    kernel_ComputeOnWorkers);
  Node *verify = new Node("verify", kernel_Verify, nullptr);

  compute->addDependency(distributeB);
  verify->addDependency(compute);

  Distributor d(argc, argv, verify);

  unsigned N = 727;
  Data B_(new int[N*N], { N, N }, sizeof(int));
  Data C_(new int[N*N], { N, N }, sizeof(int));
  Data N_(&N, { 1 }, sizeof(unsigned));

  d.addArguments({ { "B", &B_ }, { "C", &C_ }, { "N", &N_}});
  d.addOutput("Hello, world!");

  err |= d.run();

  return err;
}
```

Listing 2.1: Minimal code sample to show core functions of the library

2.1 Core Class - Distributor

The first thing the programmer will have to do is splitting the problem into different computation steps, which we call nodes (of the compute graph). After that, we set the target node of the problem, which is the last part that will be computed, and which will rely on all the previous computation steps. In the case of the minimal example in listing 2.1 we want to compute first and then *verify*. Calling *Distributor::(int argc, char** argv, Node* target)* will initialize the library and cluster, and define the target node, and so the starting point for the library to search for dependencies and start solving the problem.

Additionally, it checks for cycles in the compute graph and adds all connected nodes to an internal list. After initializing the library, all nodes and their dependencies can be inspected in the generated file *graphDependencies.png*, a DOT-graph[1] that has been generated while parsing the nodes.

All processes, master, and workers need to operate on the same graph. Control flow is triggered by barriers and data send/receive waiting, for example, workers that need to wait for data from the master in the *kernel_ DistributeB*.
Nodes will require additional arguments, in case of our sample that B matrix. Adding them as arguments to the Distributor or to single Node-objects makes them accessible within the kernels. The library stores them as a map, the programmer only needs to the remember the meaningful key he has chosen and the type of data the Data-object points to, whenever the data is requested again. More details about Data-objects are described in section 2.3.

The MPI standard does not specify in which order the output from the master process and workers will show up on the console. To avoid messy output, the library can collect each process' stdout and stderr by using the functions *addOutput(string)* on the Distributor-object (for general output) or current Node-object in the kernel for kernel-specific logging. According *getOutput()* functions can be used to send all collected logging at once to stdout or any other streamwriter.
Additionally, a silent mode has been implemented (command line argument: *silent*). Stdout and stderr will be written to streamwriters instead of the console, the master process collects these in the end when *Distributor::run()* has finished, combines them correct ordered by workers and writes the output to files named as like the executable, postfixed with ".out.master" or ".err.master". See section 2.5.5 for comments why we did not use the built-in feature of OpenMPI for that.

2.1.1 Create Cluster/Start MPI Processes

The master process will create/reuse the cluster and launch the requested MPI processes. This initialization is quite a complex process but completely hidden in the library, so that the programmer does not need to care about it. When calling the constructor of the library, it initializes the cluster, distributes the binary and starts the binary on all instances.
If no cluster exists (indicated by no file "mpi.hostfile" exists), the library starts launching a new cluster by calling an external script: *createInstances.sh*. Cluster creation is automated, still, there are various customizations, for example, the number of workers, that can be submitted as a command line argument (*W=NBR_ WORKERS*) and will be forwarded to the cluster creation script. See chapter 3 and chapter 4 for more information on how to launch a cluster.
An advanced feature is that the library dynamically starts workers and can reduce their

[1] http://www.graphviz.org/content/dot-language

number again. This is untypical for MPI and made troubles during the development.
New processes are launched by using the MPI function *MPI_ Comm_ spawn*[2]. This creates the requested number of new MPI processes on the instances like it is specified in the mpi.hostfile. The spawned processes (children) have their own *MPI_ COMM_ WORLD*, separate from the master process'. Instead, an intercommunicator between master and workers is created. This adds complexity to the handling of MPI communication, but the programmer needs not to be aware of this. The Data-object and Distributor-object hide and take care of that.
Before launching the child processes, the executable needs to be distributed to the workers - also done automatically. See section 2.5.1 for some comments about this.
After launching the processes, master and workers could start computing the problem. However, as mentioned, cluster resizing (=shrinking) is another feature of the library. If a resize has occurred, the library constructor checks if we start from scratch or continue a previous computation by searching for a checkpoint. To be able to continue the work and lose no data, all processes that remain in the cluster will write all currently used data to disk.
The programmer does not need to take care of the checkpoint or any data, everything that has been handed over to the library or nodes in form of Data-objects will be saved. Find more explanations in section 2.4, section 2.3.1 and section 2.5.2 [3].

2.1.2 Solving the Graph/Problem

A simple call to *Distributor::run()* will start computing the nodes and solving the target. Since all information about which nodes are involved and what data is required has already been set, starting the actual computation does not need much attention from the programmer.
Internally, the library starts at the target node and adds all dependencies recursively into a set. If a node has no dependencies or its dependencies are already computed, its kernel can be executed by calling the corresponding master or worker kernel. This repeats until the target node can be computed. For benchmarking and performance troubleshooting, the library tracks the start/stop of every single node.
Also, another DOT-graph *graphComputation.png* will be generated. Since the programmer can only indirectly, by setting dependencies, control in which order the problem will be solved, this might be helpful for debugging or documentation. Also, the graph contains which workers received a chunk when the master distributes work or documents where a restart has happened. Figure 2.1 shows such a dot-graph.

2.1.3 Dealing with idle/busy Workers

The functions used to distribute compute-chunks to workers are not displayed in the minimal example in listing 2.1. Instead, the kernel *kernel_ ComputeOnMaster* and the function *waitForWorker* in the provided sample matrix multiplication (*sampleMMul-simple.cpp*) shows how to use it. But the general operation will be explained here.

[2]https://www.open-mpi.org/doc/v1.10/man3/MPI_Comm_spawn.3.php

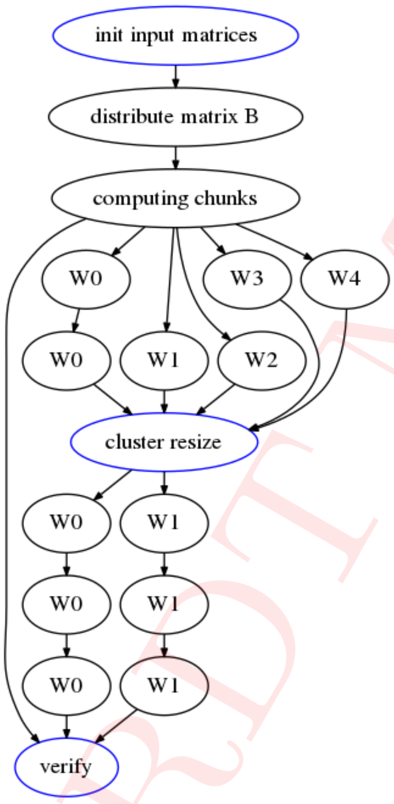


Figure 2.1: Graph for a matrix multiplication, starting with 5 workers, resized to 2

The master process will get the IDs of idle workers from the library. Dealing with workers should be asynchronous, so we probably need to remember which worker has computed what chunk of data. Once a worker has finished its computation, some data will probably be sent back to the master. This typical setup makes it necessary to split the handling of workers/chunks into different functions and different places in the code.
The library distinguishes workers by their rank. Remember that *MPI_Comm_Spawn* uses a unique communicator for all spawned workers. The first worker will have rank 0, as like as the master node. But in a different communicator, so the library can distinguish them. The programmer only needs to call *int workerID = Distributor::nextWorker(Node&)* to receive the ID of an idle worker. Peer to peer communications require an ID as receiver, this is exactly where that *workerID* should be used again.
To remember which worker has executed what chunk, a *map<int, int> worker_chunks* should be used. If you iterate over your chunks of data, store the current chunk id (=*loopCounter*) and *workerID* in your map: *worker_chunks[workerID] = loopCounter*.
If no worker is idle, *nextWorker* cannot deliver a valid worker ID. Instead, *Distributor::NO_IDLE_WORKERS_AVAILABLE* will be returned. This will imply that the master process has to wait for workers that finished their computation and collect their

data first. In a nutshell, the master will create a Data-object *result*, that is large enough to handle the expected result from any of the currently busy workers. Calling *MPI_Status status = result.recv_M_from_W()* will cause the master waiting for a result from an arbitrary worker. The *MPI_Status* object will contain the senders *workerID*. Using the previously generated map, we know which chunk that worker has computed and can fill the sub-result at the according to the position of the result-matrix C.
The master needs to tell the library that the worker is now idle again since the library has no clue if more data could be requested or the problem needs to react specifically. Calling *Distributor::addToIdleQueue(Node&, int workerID)* will mark the worker as idle again and subsequent calls of *nextWorker()* will now return a reusable worker ID. Additionally, *addToIdleQueue()* logs the current workerID in the *graphComputation.png* as receiver of a chunk.
Once all chunks have been computed, the master will continue receiving the remaining sub results. If the workers will not be needed by any subsequent nodes anymore, the master shall send a special flag to all workers by calling *Distributor::terminateWorkers()* to tell the workers that they can return from the current Node and finish the graph. Again, the library cannot know if the programmer will need the workers later or not. The library offers a possibility for the master to communicate with the workers, the programmer has the responsibility to react on the flags.

2.1.4 Stopping the Cluster, Terminating Instances, Collecting Logfiles

Finally, the Distributor can stop working. At the end of *Distributor::run()*, the library finalizes MPI and optionally shuts down the workers. This can be configured with *Distributor::setShutdownFlag(bool)*, the default is false for not shutting down workers. The master initiates the shutdown by calling *sudo shutdown -P +1*, so with one minute delay, via ssh. We have chosen this method because of the way MPI deals with child processes on termination, as explained in section 2.5.3.
Before this, the master process checks if it is restarting or terminating. In the former case, the currently used *mpi.hostfile* will be rewritten to contain only the reused instances. The superfluous instances will be shut down. See section 2.5.4 for some remarks on the method we used for the restarting.
In the latter case, all instances will be shut down. Before, the workers will send their stdout and stderr in case the Distributor has operated in silent mode (command line argument: *silent*).

2.2 Core Class - Node

A Node represents logical groups of the code and builds, in connection with the other nodes of the program, the graph of the problem that wants to be solved. Since the Distributor is responsible for the execution and control flow, the Node-class is rather simple and focuses on encapsulation.

The most important parts of a Node are the two kernels: one for the master and one for the worker. We decided to distinguish between these kernels because we think it is closer to the natural way of writing typical master/worker code. Some parts will be identical, some parts completely different, and often you will find something where the programmer requests explicitly execution by the master only.
Kernels are, as already explained, simply function pointers. The member function *Node::run(Distributor&)* will be called by the Distributor, once this current Node-object is ready to be computed. The function checks if the code currently runs on a master or worker, and calls the corresponding kernel unless that pointer is a *nullptr*. The kernel code gets access to both, the Distributor and calling Node, as arguments.
Data-objects can be bound to the Distributor or Node (*addArgument(string &, Data*)*). Syntax and behavior are identical. Same for adding log text (*addOutput(string)*). Choose which object's function you want to use depending on if you want to share the arguments with other Nodes or for output if the information is Node-specific or general information.
For benchmarking and performance checks, each Node's execution time is tracked. The Distributor offers the function *vector<pair<string, unsigned long long>> getDurationDetails()* to get these information at the end of the program.

When it comes to cluster resizing/restarting, a Node may need to provide information on its current state, like the variable *loopCounter* in the *sampleMMul.cpp*. If this variable is used, it is automatically saved in a checkpoint if the cluster is restarted. Same with Node or Distributor loglines. See the corresponding documentation of the Distributor in section 2.1.3 and section 2.3.1 for more information.

2.3 Core Class - Data

The Data-Class is a typical wrapper: MPI is a powerful library, and the Distributor uses quite some of the functionality and adds several own features. Since simplicity is one of the main goals of this thesis, and the communication between the master and workers is a bit more complicated than usual with MPI, wrapping communication into simpler functions was a logical step.
For constructing a Data-object, we need a pointer to the actual data, the number of elements and size of one element: *Data(void*, vector<unsigned>, unsigned)*. The number of elements of the original object may be multidimensional.
We decided to not use templates, but void pointers. This has the disadvantage that we need to cast the pointer each time we request the original data pointer of a Data-object and also need to additionally store the size of one element.
The advantage and main reason for our decision were, that if we would use templates, we could not store Data-objects of different types in the same map of arguments in the Distributor or Node-object, or at least we could not avoid casting. With void pointers, the map of arguments is no problem. We still need casting, but on the raw data only. Mixed types in arguments will usually be unavoidable. For example, the dimensions of a matrix multiplications will be unsigned, the data elements for example doubles.

The constructor does not copy the original data, only pointers. That makes its creation fast and using the features of the class comes with minimal computational overhead. For any changes to the data, it is important to remember that the library knows and uses these pointers. They may not be mutated. Also, additional auto parallelism might be restricted because of the use of pointers.

The master process and worker processes have separate communicators, and in between an intercommunicator. Each communication requires a communicator and in most cases also the id of a receiver. Choosing the correct objects is error-prone and it makes sense to hand this over to the library. We wrap the basic MPI functions, give meaningful names and take care of every parameter we know in advance.
For example, peer to peer communication between the master and workers has usually this signature: *int MPI_Recv(void *buf, int count, MPI_Datatype, int source, int tag, MPI_Comm, MPI_Status*).*
If we want the master to receive from a worker, we know that we need to use the intercommunicator instead of the MPI_COMM_WORLD.
The buffer and a maximum number of elements are set in the Data-objects constructor. Instead of the correct MPI_Datatype, we always use *MPI_Byte* (see section 2.5.7 for a comment on heterogeneous environments). The actual number of bytes can be calculated by the knowledge of the size of one original element, which is also known in the Data-object.
The master will often wait for just any worker's result, for example, the first one which has finished the computation. Also, we often know what data we will receive and do not need to wait for specific tags. So we can use in many cases MPI_ANY_SOURCE and MPI_ANY_TAG as default arguments, and just leave the option to use tags to the programmer.
Hence, the Data-class offers the same function with this signature:
MPI_Status recv_M_from_W(const int=MPI_ANY_SOURCE, const int=MPI_ANY_TAG).
In most cases it is sufficient to wait for data by simply calling *data.recv_M_from_W().*
The library has currently implemented wrappers for send/receive/broadcast/gather/allGather/reduce/allReduce, each for master to worker, the other way round and also for worker to worker.
Adding additional MPI message functions should be easy by extending the Data-class and orienting on the existing functions.

The master will have to deal with chunks of data. For splitting, a Data-object offers the two functions *vector<Data*> sliceSize(unsigned size)* and *vector<Data*> sliceParts(unsigned parts).* Both split the original data, the Data-object points to, into chunks.
The former will create chunks with the given size (=number of elements). If the Data-object contains a number of elements that can't be divided into chunks of size 'size' without remainder, the last chunk will be smaller and contain the remaining elements.
The latter function splits the data into a number of 'parts' parts. Again, the last chunk

will be smaller and contain the remaining elements if ‘parts’ does not divide the number of elements without remainder.
The slice functions do not copy or modify the original data, it just creates pointers to the start elements of each chunk. Hence, this operation is fast.

Tags can be freely chosen. However, there are some predefined Tags. Two of them are *RESTART_TAG* and *TERMINATE_TAG*. We expect workers to wait in an infinite loop for incoming data chunks from the master node. If the master decides to not use a worker anymore, it will send a one-byte package with a tag to this worker. Depending on if we want to resize the cluster or all work has finished, the worker has to listen and react to these tags. We expect the worker to stop and exit the Node as fast as possible, doing only the most necessary things. However, we did not dare to take over that control completely in the library, so we forward the tag to the worker, and expect the programmer to do the necessary work like storing states as Node’s Data-arguments to prepare for shutdown/restart and then exit the current Node. From that point, the library takes over control again, initiates if necessary the checkpoint and continues to the programs exit. In case of a restart, no further Nodes will be executed before the restart has been performed.

Last but not least: communication is often a bottleneck. We track all time spent on communication and sum them up. The Distributor offers a member function *Distributor::getDurationDetails()* to retrieve these timings.

2.3.1 Advanced Feature: Automatic Cluster Resizing

In case of long-running computations, it might be convenient and cost-efficient if the library would reduce the number of workers as soon as it is clear that not all workers will be required to finish the computation within the next hour[3].
To automate this, the library needs to know three things:

- The creation time of the cluster. Remember that the library launches a new cluster only if no *mpi.hostfile* has been found. Only, in this case, we know when this cluster has been created and so when the next payment-time unit will start. So do not provide an *mpi.hostfile* if you want to use this feature. Instead, the library will call the script (*createInstances.sh*), and expect it to create as a side effect the *mpi.hostfile*. Samples for creating a Virtual Box cluster or EC2 cluster are provided.

- Next, we need to know how many chunks in total have to be computed. Set this in your master compute Node by calling *Node::activateAutoResize(unsigned NBR_CHUNKS)*.

- Last, in the loop that distributes the chunks, you need to use as loop counter a variable provided by the library: *unsigned *loopCounter = Node::getLoopCounter()*.

[3]By the time of starting this thesis, Amazon EC2 instances were paid per hour, starting with their launch, ending with their termination

Provided with this information, the library will activate restarting. When we reach the end of a payment-unit (*Distributor::CLUSTER_TIME_UNIT_SECONDS*, defaulting to 3,600 seconds), the library tries to estimate the remaining computation time. It calculates the time consumed for the current number of computed chunks and extrapolates the estimated time required for the remaining chunks with the current number of workers in the cluster. If it that can be computed in less than one time-unit, the library calculates the minimum number of workers required to finish within the next time-unit, initiates a checkpoint/restart for this minimal number of workers and shut down the superfluous workers.
Since all currently running computations need to be finished first and master and some workers need to write a checkpoint, there is a static buffer on the time-unit included: *Distributor::CLUSTER_TIME_UNIT_MAINTENANCE_BUFFER*, defaulting to 0.9. This means the cluster won't be resized if the remaining work would take more than 54 minutes ($= 0.9 * 3600$ seconds). Also, the estimations won't start before the 54^{th} minute of the current time unit has been reached, since the current unit has already been started and needs to be paid, so reducing the number of workers wouldn't reduce costs.
The cluster resize estimation is triggered when workers become idle, when the master calls *Distributor::addToIdleQueue(Node&, int workerID)*.

2.4 Advanced Feature: Core Class - Checkpoint

In case a restart is done, all affected workers and the master will write their current state to disk, to a file *checkpoint.sav* in the same directory like the executable (on workers: */tmp*).
On the master, the library has control when the restart happens. The decision if a restart is made is triggered by adding former busy workers to the list of idle workers again (*Distributor::addToIdleQueue()*). If we are restarting, the library does not release idle workers anymore, forcing the master to continue dealing with the remaining results of the currently busy workers until all workers have finished their last chunk. Then, all workers will be idle, the master compute kernel nevertheless won't receive idle workers and is forced to stop the current Nodes' loop over the workload. The checkpoint will be triggered automatically by the last worker that has finished its work. Because of that, the master process needs to stop working in the current node immediately, because all work this process does won't be included in the checkpoint and be lost once the restart has been performed.
For the workers, we do not have that much control and require the programmer to stop working and call *Distributor::.saveCheckpoint(Node&)* themselves. Again, after writing the checkpoint, all future work by that process won't be included in the checkpoint and be lost after the restart.
The checkpoint contains:

- the current state of the DOT-graph
- current stdout/stderr in case of silent-mode

- all timings from the Distributor
- all timings from all Nodes
- all timings from all Data-objects
- all arguments from the Distributor
- all arguments from all Nodes
- the finish flag of all Nodes
- loop counter and number of total chunks for all Nodes (if defined)

The Nodes' description and dependencies do not need to be saved. Remember that master and workers create these objects before the cluster is initialized. So, each process creates a new graph from scratch but modifies the Nodes with the data from the checkpoint when we restore it. The following call *Distributor::run()* will now know which Nodes previously have been computed and can continue the work where we have left by restoring the previous loop counter and interim data from all arguments.
The checkpoint will be restored during the initialization of the cluster. After restoring, the checkpoint file will be deleted again.

2.5 Problems and Solutions

OpenMPI has many nice features. Sadly, we have made the experience that if you use less common features together, it seems that the versions we have used behave a bit strange. Perhaps we have found by accident one or another bug, or did not recognize that some of the features may not be combined.
Because of that, quite often we needed to remove MPI features we were using for some time again, and find alternatives respectively have written code for that features by ourselves. We had no intention to reinvent already available features, but in many cases, we did not find a working solution or a solution that seemed to run stable on all the different combinations of MPI versions and operating systems we have used during the development.

2.5.1 Distributing the Binary

In theory, OpenMPI provides a feature for this, *−−preload-binary*[4]. Unfortunately, with OpenMPI version 1.10.2, this feature became buggy when used in combination with *MPI_Comm_Spawn*. The binary has been distributed, but the execution bit has not been set, and so MPI could not execute it. This feature worked fine with OpenMPI v1.6.5., which is the default version the package manager offers for Ubuntu 14.04 and Debian 8. So, since MPI needs passwordless ssh anyway, we stopped using this feature

[4] www.open-mpi.org/doc/v1.10/man1/mpirun.1.php

and have written our own method. The function *make distribute* has been added to the Makefile, it parses the *mpi.hostfile* for all used instances and transfers the executable using SCP to */tmp/*.

2.5.2 Cluster Resize - Loosing MPI Child Processes

MPI has not been designed to reduce the number of MPI processes. You can remove processes from communicators and groups and stop sending data to these processes, but if the unused worker loses its connection with the master process, the whole application crashes. Since we want to work cost-effective on Amazon EC2, we need to shut down idle workers and cannot keep these MPI processes alive.
To work around this problem, the only solution that worked on different MPI versions and operating system combinations that have been found is stopping all MPI processes, shutting down the now unused workers and restarting the master with a reduced number of workers again.
We have spent a lot of time finding a better solution for this feature. Losing MPI processes did not make any problems using OpenMPI v1.6.5 on Debian 8. But it did not work with the same MPI version on Ubuntu 14.04 or with OpenMPI v1.6.4 on the Amazon GPU instances, which is our main target for using the resize-feature to save costs.
MPICH had a promising feature called *-disable-auto-cleanup*. Unfortunately, it did not help continuing with the computations if unused workers are shut down. Once the master process loses a child MPI process, the whole application will be killed.
So, following the few pieces of advice that can be found on the net talking about the loss of MPI processes, we decided to completely shut down the application and restart with the reduced number of workers.

2.5.3 Terminate Workers

MPI kills child processes if the main program terminates. It seems this is not implemented in all versions of OpenMPI in the same strict way. In an early stage of development, we did not notice problems when we let the workers initiate their own shutdown by using *system()* and a delayed execution of *shutdown*. By that time we were mainly using OpenMPI v1.6.5 on Debian 8 and Ubuntu 14.04.
After migrating to Ubuntu 16.04/OpenMPI v1.10.2, as soon as the main program has terminated, MPI seemed to kill also all child processes on the worker, including our attempt to shut down. Switching from *system()* to other methods like *fork* or similar did not help.
We helped ourselves by calling shutdown from the master node via ssh. Side note: calling shutdown via ssh on to worker to itself also did not work - the shutdown process was simply killed when mpirun stopped the execution of the main program.

2.5.4 Restart the Cluster

As mentioned before, MPI does not allow losing processes before the master process has finished. Also, once the master node finishes, all child processes are terminated. So it is also not possible to simply start a new instance of the program with different parameters or on a new version of the mpi.hostfile (error message: "Open MPI does not support recursive calls of mpirun"), and the master can also not start itself again before termination. That worked only when the master had not been started as an argument of mpirun. But unfortunately, not using mpirun for the master process came up with errors when calling *MPI_Comm_Spawn*, when we wanted to initialize workers on the same instance on which the master process runs. We wanted to do this in case we are running on a GPU cluster, and one server should additionally run the master instance instead of wasting this instance's GPUs by not launching any worker on that instance.

So, once again we fixed this by using a very cruel method: the restarting master writes a new mpi.hostfile with the minimal number of workers. Then, it writes a check file (*restarting*) that informs the shell that has started the master process, that it should start the master process after termination again.
Usually, if the library can estimate the remaining computation time correctly, only one restart should be required. However, we have written the routine that starts the master process to call it subsequently, as long as this check file *restarting* exists.
Therefore, we recommend that the programmer uses similar scripts or reuses the corresponding function in the Makefiles of our samples: *make run* or *make runSilent*.

2.5.5 Redirecting output

The MPI standard does not define the order in which MPI processes write to the console. In practice, the output is most often difficult to read because it is a messy mix of all streams. OpenMPI has a feature −−*output-filename*, that instructs all MPI processes to pipe their output into a text file instead of the console. This works unfortunately not in combination with *MPI_Comm_Spawn*, at least not in all OpenMPI versions. Once again, this feature worked on version 1.6.5, but not with 1.10.2.
So we got rid of it and implemented it by ourselves, using a redirection for stdout/stderr on demand and sending the workers output to the master right before a worker is going to be shut down. To activate this feature, add *silent* as a command line parameter or call the program with *make runSilent*.

2.5.6 Pros and Cons of calling the master with "mpirun"

Usually, MPI programs are started by calling mpirun and adding the executable as an argument. However, you can also just call the executable, if don't need to immediately call MPI functions like starting remote processes. Since we delay the start of processes, we could start our application without prefixing the command with mpirun and MPI arguments.

We noticed, that it has advantages and disadvantages for us to prefix our application start with mpirun. If we do not use mpirun, we can have our master start itself again for cluster restart. If we use mpirun, we get an error when we try to launch another master process, error message: “Open MPI does not support recursive calls of mpirun”.
Also, child processes like the termination of workers won’t be killed by the wrapping mpirun if you don’t call the master process prefixed with mpirun.
But if we do not use mpirun, we noticed a very strange behavior on spawning processes on the same instance, on which the master runs. Instead of just adding the requested number of workers, we noticed that either the master has been oversubscribed and some instances where missing workers, or the program terminated with a message that not enough slots were available.
We believe that we have observed a bug since the behavior seemed not to follow a system and did not make any sense or acted like the descriptions in the official documentation said how it should behave.

2.5.7 Using void* and MPI_Datatype

Since we work with void pointers in the Data-class, we cannot easily figure out with what data type we are currently dealing with. But we know the size of it and so can send/receive it as raw data (MPI_Byte). The data will be restored to their original type by casting them back. This is a necessary, but easy task we leave to the programmer’s responsibility.
However, in case we transfer data between systems with different configurations, for example, different operating systems or between 32-bit and 64-bit versions, the same data type will not necessarily have the same size.
We did not make tests with such different systems and cross-compiled binaries for the master and workers. Also, MPI requires the same version on all members of the cluster, so working on heterogeneous clusters will always be a problem. But in case this is an important feature, keep in mind that every Data-object works on bytes and check for yourself that they will be interpreted correctly between instances with different environments.

3 Amazon EC2 – Cluster

Using GPU instances, offered by cloud providers, is an interesting option for people who cannot or don't like to host their own GPGPU cluster in their basement. As long as you do not need the servers for too many hours on average per day, this might be a huge cost reduction.
By the time we have written the proposal for this thesis, Amazon EC2 was first offering GPU instances. We did not find reports of successful implementations in the field, so we added the use of EC2 as a goal and testing the GPU instances for their efficiency as a prerequisite, described in section 3.1. Some instances alone do not make a cluster, the necessary tasks to get our code running on the servers are described in section 3.2, 3.3 and 3.4.
Before we start, find in table 3.1 a list that describes the different available instance types and their expansions for an overview:

g2.2xlarge:	1 GPU	8 CPU cores	15GB RAM	\$ 0.65/Hour
g2.8xlarge:	4 GPUs	32 CPU cores	60GB RAM	\$ 2.60/Hour
g3.4xlarge:	1 GPU	16 CPU cores	122GB RAM	\$ 1.14/Hour
g3.8xlarge:	2 GPUs	32 CPU cores	244GB RAM	\$ 2.28/Hour
g3.16xlarge:	4 GPUs	64 CPU cores	488GB RAM	\$ 4.56/Hour
p2.xlarge:	1 GPU	4 CPU cores	61GB RAM	\$ 0.90/Hour
p2.8xlarge:	8 GPUs	32 CPU cores	488GB RAM	\$ 7.20/Hour
p2.16xlarge:	16 GPUs	64 CPU cores	732GB RAM	\$ 14.40/Hour

Table 3.1: Overview Amazon EC2 instances types as of 07/2017

All prices represent the region in North Virginia, US, in July 2017. They only serve as a guideline. For current pricing, please visit the Amazon EC2 web page[1].
We did our tests on the g2-instances. Contracts between our University and Amazon included use of them but not the later introduced p2 or g3 instances. We believe that our code will work on the these instances and other upcoming types as well. Perhaps some minor changes for driver and library paths might be necessary, but since this is daily business when dealing with product life cycles we are not concerned about that.

The g2-instances offers up to 4 NVIDIA GRID K520 GPUs, with 1,536 CUDA cores per GPU. They are optimized for single precision floating point operations, in contrast to the more powerful GPUs used for the more recently introduced p2-instances. On the

[1]https://aws.amazon.com/ec2/pricing/

other hand, they are cheaper, so be aware of what kind of GPU you need if you want to choose the most cost-efficient server.
The g3-instances were announced in July 2017 and are optimized for graphics-intensive applications. They use NVIDIA Tesla M60 GPUs (2,048 CUDA cores) and Intel Xeon E5-2686 v4 processors.
P3-instances offer NVIDIA K80 GPUs (2,496 CUDA cores), which are optimized for GPGPU and especially more powerful when it comes to double precision floating point operations [1], [2].

For network communication, EC2 offers the possibility to get instances that are connected with a fast 20Gbps interface, if they are put in a so-called "placement group". In our implementation, we use this feature by default. If Amazon is short on instances, it might be impossible to deliver a number of instances that can be put into one placement group, but possible to get this number spread over different, less fast connected locations [2].

3.1 Benchmark and Comparison

Since we have found no representative reports other than from Amazon or NVIDIA about the expecting performance at the time we defined the thesis, we decided to run a few benchmarks and compare an EC2 instance with a stand-alone Tesla K20 from the laboratory in our University. We were interested in, if the server might lose performance because of the virtualization layer or simultaneous use of different GPUs on the same host, by other users.
As benchmarks, we have chosen programs that we have written as part of a former project. We are running three tests that differ in their hardware requirements, but are typical algorithms that might be used for GPGPU:

The first algorithm is a simple micro benchmark. Some common math-functions are tested, with minor differences for data types integer, float and double.
Second is a PI-calculation using the Leibniz-formula[2]. The calculation steps are simple and do not differ enough from the micro benchmark to deliver interesting results, but the sub-results have to be summed up for the final result (reduction) which is a frequently used operation in GPGPU.
Third is a matrix multiplication (SAXPY operations), executed in three different ways: Naïve with swapped rows/columns by intention to provoke higher demand on memory accesses, called "stupid", next the straightforward, "naïve" algorithm everyone knows from linear algebra and a more sophisticated method using "tiling", which is more memory efficient.

For variety and possible additional differences in performance, not only single precision (float) has been measured, but also double (DAXPY) and integer (IAXPY) for all

[2]https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80

three benchmarks. Just the Leibniz algorithm works with floats and doubles only since it would not make sense to compute Pi with integers.

Benchmark	int	float	double	average
Leibniz	n/a	0.71	0.24	0.48
Microbenchmark	1.37	1.10	0.14	0.87
MMul (stupid)	0.80	0.80	0.82	0.81
MMul (naive)	1.03	0.93	0.72	0.89
MMul (tiling)	1.10	1.18	0.82	1.03
average	1.08	1.00	0.62	**0.84**

Table 3.2: Benchmark comparison: EC2-g2 NVIDIA GRID K520 vs. Tesla K20

Table 3.2 shows the computation time of the benchmarks on an Amazon g2-instance, divided by the computation time on the stand-alone Tesla K20. We observe some slightly better performance for the naïve integer and tiling integer matrix multiplication as well as for the float matrix multiplication. Also, for the micro benchmark, we see better results for the g2-instance. When it comes to double precision, the performance drops. This was expected. As shown in table 3.3 NVIDIA's documentation stated a higher memory bandwidth for the Grid K520 (which is the same as Tesla K10 but modified for use in multi GPU servers and it has a slightly increased core clock speed). On the other hand, the K20 has a much higher peak double precision floating point performance, 1.17 Tflops versus 0.19 Tflops, which explains the weakness when it comes to doubles precision [6], [5], [7].

	Tesla K20	Grid K520
Peak double-precision	1.17 Tflops	0.19 Tflops
Peak single-precision	3.52 Tflops	4.58 Tflops
GPU	GK110	GK104s
Number of CUDA cores	2,496	1,536
Memory size	5 GB	4 GB
Memory bandwidth	208 GBytes/sec	320 Gbytes/sec

Table 3.3: Technical specifications: Tesla K20 vs. EC2-g2 NVIDIA GRID K520

Our benchmarks have not the intention to measure if the K520 in the g2-instance performs at its maximum achievable power. We wanted to compare it with some other GPU, that might be used for an on-site GPU cluster. In a nutshell, the GPU in a g2.2xlarge instance performs a bit slower, at about 0.84 percent of the power of a Tesla

K20 in a non-virtual server as stand-alone hardware. This might be a helpful information when someone needs to evaluate the computation time and costs spent on EC2 compared with a self-maintained GPGPU server infrastructure.

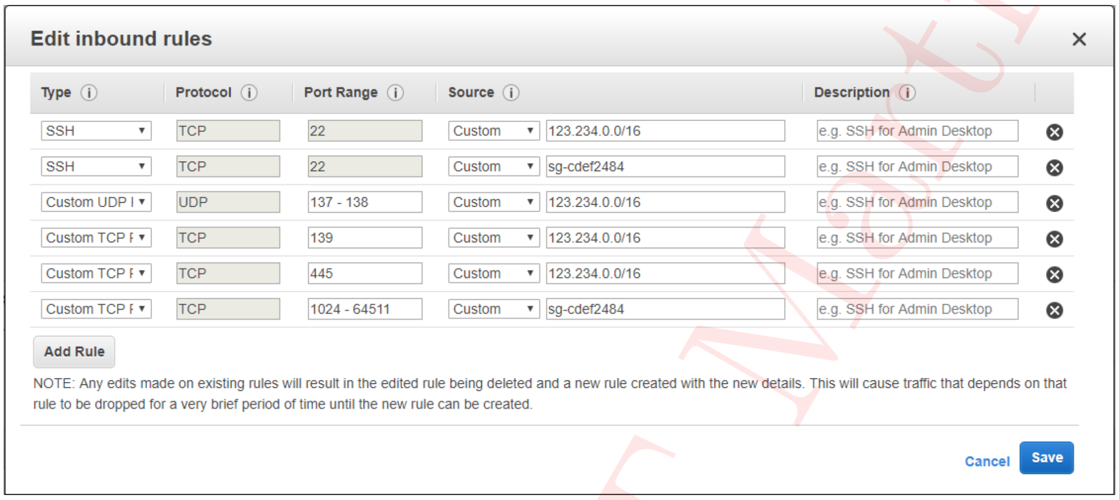
3.2 Configuration of AWS-Cli, AWS-EC2 and AWS-IAM (initialConfig.sh)

Before creating a cluster, we need to configure your local machine and install some tools. Therefore, we have written the script *setupDistributedGPGPU.sh*, which has been tested with Ubuntu 16.04 and Debian 9. Modifications for installing packages on other Linux distributions should not be a big problem. The script assists in installing prerequisites for the library and for the Virtual Box test environment (see chapter 4) as well. The last three installation steps install and configure the EC2 CLI and environment: installing tools for EC2, configuring EC2 and parsing for the most current Linux images on EC2.

If you want to install the AWS CLI only, you can skip all steps until "setup AWS console". Since we need to install some packages, it will ask you for your sudo password. Please have sudo already installed or execute the necessary parts of the script with your preferred method. We need *python-pip*, *awscli*, *dnsutils (dig)* and *jq*. Once they are installed, we configure the AWS CLI, and therefore we need your “AWS Access Key ID” and “AWS Secret Access Key”. You get both when you create your Amazon IAM user. Find details to these keys in the AWS online documentation[3]. Additionally, you can specify a default region to create your instances into, for example “us-east-1”. The selected region has impacts on the costs of AWS instances and of course on the latency between your computer and the cluster. As already mentioned, we use homogeneous clusters only. For that reason, your on-site machine will probably only be used to connect to a cluster, upload the input data and code, compile remote and wait until the computations have been completed. So, latency might not be a major concern, choosing a cheap region could be more interesting.

The configuration affects two parts: locally, the user credentials, and remote, on the AWS administration servers, some settings for the cluster environment.
The remote configuration concerns general firewall settings (cloud configuration bound to the EC2 user, nothing local) for the AWS cluster, as shown in figure 3.1, like external ssh/smb/NetBT access, and creates a user with restricted permissions, that is only allowed to create g2 or t2.micro instances (g2 is the typical GPU instance, t2 a CPU only instance, which is much cheaper and can be used for tests).
We need this user on the master node (see chapter 3.4), and since for real-world computations this will be an EC2 instance, we don't like to store personal AWS user credentials on that off-site instance.
With this user/key pair, you can't login to the AWS console, and we used a policy

[3]http://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html



Type	Protocol	Port Range	Source		Description
SSH	TCP	22	Custom	123.234.0.0/16	e.g. SSH for Admin Desktop
SSH	TCP	22	Custom	sg-cdef2484	e.g. SSH for Admin Desktop
Custom UDP I	UDP	137 - 138	Custom	123.234.0.0/16	e.g. SSH for Admin Desktop
Custom TCP F	TCP	139	Custom	123.234.0.0/16	e.g. SSH for Admin Desktop
Custom TCP F	TCP	445	Custom	123.234.0.0/16	e.g. SSH for Admin Desktop
Custom TCP F	TCP	1024 - 64511	Custom	sg-cdef2484	e.g. SSH for Admin Desktop

NOTE: Any edits made on existing rules will result in the edited rule being deleted and a new rule created with the new details. This will cause traffic that depends on that rule to be dropped for a very brief period of time until the new rule can be created.

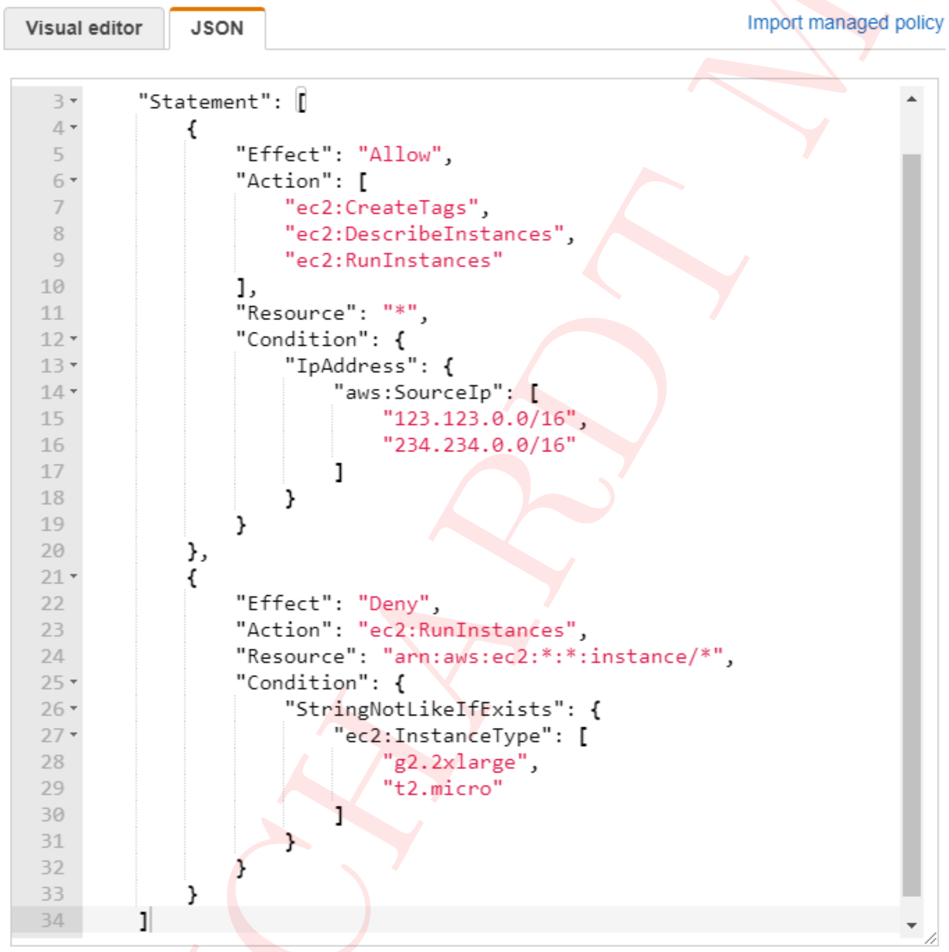
Figure 3.1: Sample firewall settings for restricted inbound ssh/smb/NetBT access

(*setup/templates/policy.ec2-master*, see figure 3.2) that allows this user to only launch instances and modify their states, and only g2.2xlarge as well as t2.micro instances. Still, this is enough to not want to get this key compromised.
Side note: if you need to launch other instance types like p2 or other g2 types, please modify the policy accordingly.

However, if this second key gets compromised, you can log on to the AWS IAM console[4] and delete the user, as well as the key pair in the AWS EC2 console[5]. Your personal account will not be affected. In case you have any suspicion that the *EC2_ CLUSTER_ USER*-key might have been compromised, do not hesitate to delete it.

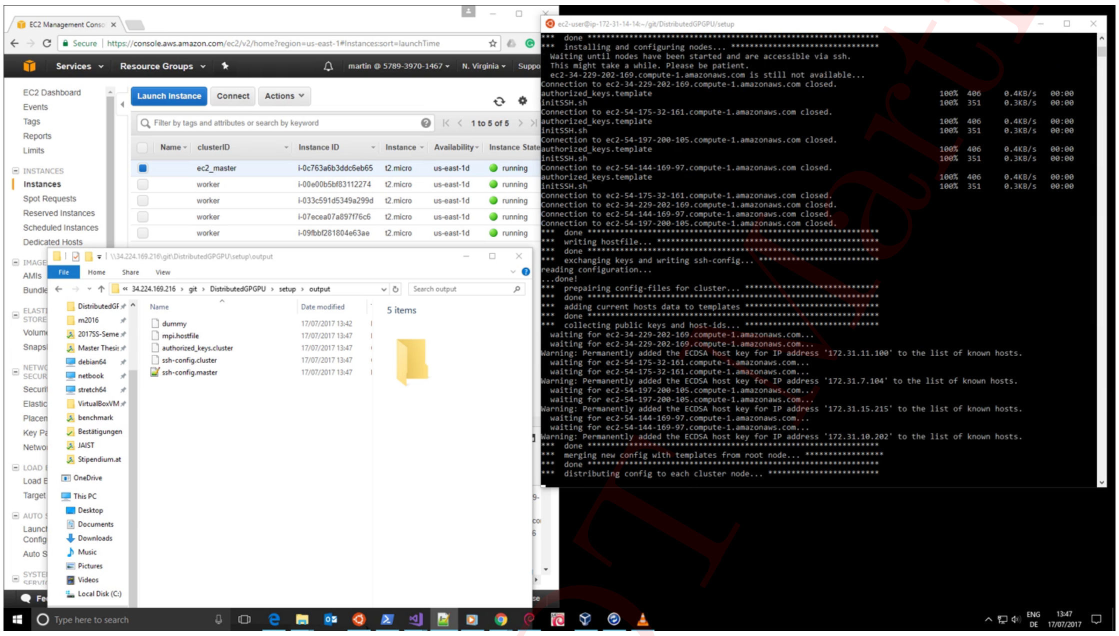
Next, we set up a placement group and search for the correct Linux image. The g2 and t2 images for AWS instances are frequently updated. Also, their identifications differ between regions. Therefore, the script searches for the most recent image in the selected region and stores it in the local configuration file (*config.cfg*).
No AWS cluster will be created now. This is done by calling *awsCreateInstances.sh keys/mpi-us-east-1.pem mpi-us-east-1 5* as an example, assuming that your generated key file (and identical key name) is *"mpi-us-east-1"* and you want to create 5 instances. The complete process can be automated, see the script *setup/demo-ec2.sh* and the corresponding video *(video/demo-ec2-master and workers.mp4)*, which shows the creation of an EC2 master instance, three workers that are created from that node, and last it compiles and tests the library, all together in 10 minutes (screenshot: figure 3.3).

[4]https://console.aws.amazon.com/iam
[5]https://console.aws.amazon.com/ec2



```
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateTags",
                "ec2:DescribeInstances",
                "ec2:RunInstances"
            ],
            "Resource": "*",
            "Condition": {
                "IpAddress": {
                    "aws:SourceIp": [
                        "123.123.0.0/16",
                        "234.234.0.0/16"
                    ]
                }
            }
        },
        {
            "Effect": "Deny",
            "Action": "ec2:RunInstances",
            "Resource": "arn:aws:ec2:*:*:instance/*",
            "Condition": {
                "StringNotLikeIfExists": {
                    "ec2:InstanceType": [
                        "g2.2xlarge",
                        "t2.micro"
                    ]
                }
            }
        }
    ]
```

Figure 3.2: Sample policy for the restricted EC2 user



https://console.aws.amazon.com/ec2/v2/home?region=us-east-1#Instances:sort=launchTime

Name	clusterID	Instance ID	Instance	Availability	Instance State
	ec2_master	i-0c763a6b3ddc6eb65	t2.micro	us-east-1d	running
	worker	i-00e00b5bf83112274	t2.micro	us-east-1d	running
	worker	i-033c591d5349a299d	t2.micro	us-east-1d	running
	worker	i-07ecea07a897f76c6	t2.micro	us-east-1d	running
	worker	i-09fbbf281804e63ae	t2.micro	us-east-1d	running

\\34.224.169.216\git\DistributedGPGPU\setup\output

Name	Date modified
dummy	17/07/2017 13:42
mpi.hostfile	17/07/2017 13:47
authorized_keys.cluster	17/07/2017 13:47
ssh-config.cluster	17/07/2017 13:47
ssh-config.master	17/07/2017 13:47

ec2-user@ip-172-31-14-14:~/git/DistributedGPGPU/setup

```
***  done ***************************************************************
***  installing and configuring nodes... ********************************
  Waiting until nodes have been started and are accessible via ssh.
  This might take a while. Please be patient.
  ec2-34-229-202-169.compute-1.amazonaws.com is still not available...
Connection to ec2-34-229-202-169.compute-1.amazonaws.com closed.
authorized_keys.template                                                  100%  406     0.4KB/s   00:00
initSSH.sh                                                                100%  351     0.3KB/s   00:00
Connection to ec2-54-175-32-161.compute-1.amazonaws.com closed.
authorized_keys.template                                                  100%  406     0.4KB/s   00:00
initSSH.sh                                                                100%  351     0.3KB/s   00:00
Connection to ec2-54-197-200-105.compute-1.amazonaws.com closed.
authorized_keys.template                                                  100%  406     0.4KB/s   00:00
initSSH.sh                                                                100%  351     0.3KB/s   00:00
Connection to ec2-54-144-169-97.compute-1.amazonaws.com closed.
authorized_keys.template                                                  100%  406     0.4KB/s   00:00
initSSH.sh                                                                100%  351     0.3KB/s   00:00
Connection to ec2-54-175-32-161.compute-1.amazonaws.com closed.
Connection to ec2-34-229-202-169.compute-1.amazonaws.com closed.
Connection to ec2-54-144-169-97.compute-1.amazonaws.com closed.
Connection to ec2-54-197-200-105.compute-1.amazonaws.com closed.
***  done ***************************************************************
***  writing hostfile... *************************************************
***  done ***************************************************************
***  exchanging keys and writing ssh-config... ***************************
reading configuration...
...done!
***  prepairing config-files for cluster... *****************************
***  done ***************************************************************
***  adding current hosts data to templates *****************************
***  done ***************************************************************
***  collecting public keys and host-ids... *****************************
  waiting for ec2-34-229-202-169.compute-1.amazonaws.com...
  waiting for ec2-34-229-202-169.compute-1.amazonaws.com...
Warning: Permanently added the ECDSA host key for IP address '172.31.11.100' to the list of known hosts.
  waiting for ec2-54-175-32-161.compute-1.amazonaws.com...
  waiting for ec2-54-175-32-161.compute-1.amazonaws.com...
Warning: Permanently added the ECDSA host key for IP address '172.31.7.104' to the list of known hosts.
  waiting for ec2-54-197-200-105.compute-1.amazonaws.com...
  waiting for ec2-54-197-200-105.compute-1.amazonaws.com...
Warning: Permanently added the ECDSA host key for IP address '172.31.15.215' to the list of known hosts.
  waiting for ec2-54-144-169-97.compute-1.amazonaws.com...
  waiting for ec2-54-144-169-97.compute-1.amazonaws.com...
Warning: Permanently added the ECDSA host key for IP address '172.31.10.202' to the list of known hosts.
***  done ***************************************************************
***  merging new config with templates from root node... ******************
***  done ***************************************************************
***  distributing config to each cluster node... ************************
```

Figure 3.3: GPGPU Cluster creation and library test on Amazon EC2

Other configuration switches from the *config.cfg* file are:

- SHUTDOWN_BEHAVIOR: *[terminate|stop]* choose if an instance should be terminated or only stopped when it is shut down. To save money, the machine needs to be terminated, otherwise, you continue to pay for your storage. But your data will be lost once the machine is terminated.

- DRY_RUN: *[−−dry-run|−−no-dry-run]* to test if you have the necessary permissions to execute an AWS CLI command, use this flag. If set, the script will not create instances and soon fail, but you may do a basic test run.

- EC2_CLUSTER_USER, EC2_SECURITY_GROUP, EC2_PLACEMENT_GROUP, EC2_CLUSTER_GROUP, EC2_POLICY_NAME: choose names for the groups, users, and policy, for example, "mpi".
  Used by the configuration step in *setup/setupDistributedGPGPU.sh* and *awsCreateInstances.sh*. If you change something, you need to run setup again.

- *REGION*: the default region you want your instances to be created into, for example "us-east-1". If you want to change your region, reset the run setup again to detect new, valid Linux images and register a new restricted user for that region.

Regarding security, we decided to restrict all connections to our cluster instances to allow only ssh and MPI within the instances and SSH/Smb/NetBT from your external IP/Subnet (as detected by textitsetup/setupDistributedGPGPU.sh). But for MPI we used the default ports, which are 1024-64511. Since we accept inbound traffic only from our created instances, not outside, we decided that it is better to use defaults than to restrict these ports.

Be aware that the configuration files *config.cfg*, *setup/keys/*.pem* and *~/.aws/credentials* contain sensible data. Do not share them with others and protect them.

3.3 Configuration of Instances (awsCreateInstances.sh)

Before the automatic cluster creation and configuration, delivered with the library, can be used, some more basic configuration needs to be done. The file *setup/config.cfg* contains information which types of instances to be created. The following settings are mandatory and have been set by the setup script:

- *NUM_ GPUS*: the number GPUs that one instance has (1 for g2.2xlarge). The *mpi.hostfile* will be configured based on this number. You can overload your instances and modify later your *mpi.hostfile* if you want to change the value or if you want to use other instance types other than g2.2xlarge with more GPUs.
- *INSTANCE_ TYPE*: for example "t2.micro" for cheap non-GPU instances for tests or "g2.2xlarge" (default) for the basic GPU instance. Comment/uncomment the corresponding lines in the configuration file to switch instance types.
- *IMAGE_ ID*: the exact version of the image you want to launch. The names vary with their version, instance type and region[6]. Examples for a t2.micro image in us-east-1 are "ami-c58c1dd3", for a g2.2xlarge image "ami-61e27177".
  Be aware that these images are often updated and will not be kept forever.
- *AWS_ ACCESS_ KEY_ ID*, *AWS_ SECRET_ ACCESS_ KEY*: the key with reduced permissions for EC2-master instances, as explained in section 3.2.

By default, EC2 customers cannot create GPU instances. In the dashboard of the EC2 console, a section "Limits" indicated how many instances you are allowed to launch. Request a limit increase if the allowed number does not match your needs.

Except for the initial connection, we will use ssh with your accounts RSA-key to communicate with the cluster instances.

The actual cluster creation is done with the script *awsCreateInstances.sh*. The first parameters specify the path to the pem-file and name of the key pair (usually identical with the name of the pem-file without file suffix), and the number of instances to be created.
All following parameters are optional, and used to create a master instance or configure

[6]https://aws.amazon.com/marketplace

already created instances. Specify that you want to create a master node by using exactly the string "master". If so, you also need to specify a password (for the samba share that will be created on the master instance) as parameter 5 or the script will read the input from the console.
Parameter 6 specifies a tag that will be shown in the AWS console and can be used for queries/filters in AWS CLI commands but is optional. If you submit this or the following parameter, you also need to submit your samba-password as a parameter as well.
Last, you can optionally enter a list of comma-separated public DNS names of already created instances, which will then be configured only, and the instance creation step will be skipped.

Creating instances works as follows:

1. create the instances (*aws ec2 run-instances*)
2. set tags (default or parameter 6)
3. wait for EC2 until all instances have been assigned a public IP/DNS name
4. collect your public RSA key for passwordless ssh (required by MPI)
5. wait for EC2 until all instances are reachable via ssh
6. copy some scripts and config to each instance
7. configure paths
8. install OpenMPI
9. install the Intel OpenCL SDK (t2.micro instances only)
10. create an RSA-keypair
11. create a *mpi.hostfile* using *NUM_ GPUS*. The master instance can be used to host workers too. To use this feature, remove the comment from the second line in the *mpi.hostfile*, which describes the master instance.
12. calling *configureAccess.sh* to exchange keys between the cluster and master instance
13. master instance only: add a rule to the firewall be allowed to access the cluster subnet
14. master instance only: copy *mpi.hostfile* to master instance

Steps 7-10 will be executed in parallel for all instances.
For step 9, you need to provide the installer and silent install file for the Intel OpenCL

SDK[7]. We have used "opencl_runtime_16.1.2_x64_rh_6.4.0.37.tgz". It is not the officially supported, but since we need it only on t2.micro instances for testing purposes, we did not worry about that. For the g2-instances, we are using the NVIDIA OpenCL implementation provided by NVIDIA/Amazon. We recommend providing a URL where the t2.micro-instances can download the files for the SDK, to avoid that your workstation needs to upload the 112MB for each instance. Configure this URL as *OPENCL_INSTALL_REDHAT* and the URL for the silent install file (see the one we provide in template/opencl_runtime_silent.cfg) as *OPENCL_SILENT_INSTALL* in the configuration file (*config.cfg*).

To use the master instance to host workers, the *mpi.hostfile* generated in step 11 just needs to list the master instance too. Since on this machine already one MPI process is running (the master process), we need to specify the number of slots by one greater than all other workers. Just uncomment line 2, this has already been pre-configured.

Script *configureAccess.sh* from step 12 is reused for creating Virtual Box clusters for a test environment, too (see chapter 4).
First, it connects to all instances and collects their public RSA keys, internal IP, external IP, and hostname. The public key will be added to all instances' and your workstations *~/.ssh/authorized_keys* file. The IP and hostname will be used to create a *~/.ssh/config* file. Depending on if we communicate within or outside the cluster, external or internal IPs will be used. The cluster instances entries for the ssh config will be added to your workstations *~/.ssh/config* as well.
Additionally, it parses your machines external IP address and ssh-server port. This information, your public key and the config and public keys of all instances will be distributed to all instances ssh-config and ssh-authorized-keys file. Once everything has been distributed everywhere, all instances can passwordless use ssh within each other, which is a prerequisite for using MPI.

If you want to use your workstation as a master instance, please read the next section 3.4 concerning firewall restrictions on incoming traffic.

3.4 Special Case - EC2 Master Instance

It would be convenient to use a local workstation or server as a master instance, distribute chunks of data to the workers on EC2 to compute, collect the results and terminate the cluster once all work has been done. Data would be on-site and modifications to the program could be done comfortably if the developer's workstation could be used as a master node.
In theory, this is no problem. In practice, you need to ensure that you have a stable connection to all your workers. MPI is unforgivable in losing connection to other MPI

[7]https://software.intel.com/en-us/intel-opencl

processes and does almost nothing regarding fault tolerance. Also, you need to receive connections on ssh and MPI ports, which default to 1024-65535. MPI can be restricted to use a smaller port range, but still, you will have some incoming traffic and we expect that many network administrators won't like that. Also, ssh is a popular target for attacks. Keeping a machine secure to safely establish connections between the Internet and Intranet is not easy and for sure no task for a developers workstation.

Since our master instance needs to communicate with our workers, we think a safe solution is to have this machine on EC2 too and connect to this machine via ssh from the programmer's network outbound only. To add some comfort to the master node and get the code working, we need additional tools, like compilers (g++), git, make, samba and prerequisites for our *awsCreateInstances.sh* script.
To configure a master instance, run *"./awsCreateInstances.sh keys/usWestNVirginia.pem usWestNVirginia 1 master PASSWORD OPTIONAL_ UNIQUE_ TAG"*. The password provided will be used for the samba server, which shares by default the folder *~/git*. The optional tag makes sense to easily distinguish the master instance in the AWS console, in case you want to manually terminate instances.

For the master instance, we loosen our security restrictions and allow ports required by samba, but only for the specified subnet that has been specified in the setup script. Once your master is created, you can connect to this share, the required syntax including your master instances' DNS name and the username on EC2 will be displayed in the output of *awsCreateInstances.sh*.

We believe this is a usable solution for all who have concerns about incoming traffic. For development, you can host a cluster on your workstation, as explained in chapter 4.

4 Test Environment - VirtualBox Cluster (vboxCreateInstances.sh)

Our main target for computations are GPGPU instances on Amazon EC2. However, for convenience and saving money we implemented a test environment using virtualized servers running on VirtualBox. The main use was for testing code that has implemented our library, but we also wanted to use our test environment to simulate the launch and configuration of instances as like we do it on EC2. The scripts we have written are adjusted to match our environment. You will probably use other operating systems, configurations, tools. Please take this chapter as a help how to create a good testenvironment for you, not as rules how you have to implement it.
For a quick start, you can use the Debian 9 or Ubuntu 16.04 based images we have created. Find the links in the repository (*setup/config.template.cfg*).

4.1 Linux Template Installation

The starting point is a scratch Linux installation, with a fixed hostname (eg. "clusterTemplate") and a network interface connected to the Internet. This two information plus the root password will later be used and modified when creating a specific instance. Next, we installed the basic drivers, SDKs, and tools that we might need for development. Also, we added permissions to shut down an instance for the user we indent to use for code execution in the sudoers file. This image is now similar to the environment we have on a g2-instance on Amazon. Except, we decided to also install OpenMPI already in that image. We used Linux Debian 9 and Ubuntu 16.04 server for our cluster templates, the scripts we used to install and configure the basic components can be found in the folders *setup/files-[ubuntu|debian]/setup[Ubuntu|Debian]Template.sh.*
Instead of cloning the base image for each cluster instance, we are using one base image and snapshots for each cluster instance's specific changes (copy-on-write), as described by Ferry Boender[1] [4]. The base images of our templates have about 4-5GB (approximately 2GB can be saved if you do not need the CUDA-SDK), the differential snapshots start with a few MB. This and the low requirements on CPU and memory allows hosting a quite high number of instances, even on weaker PCs or notebooks. We were successfully working with a 5 instances cluster on a three-year-old netbook for less than EUR 250 original price.

[1]https://www.electricmonk.nl/log/2011/09/24/multiple-virtualbox-vms-using-one-base-image-copy-on-write/

For creating the instances, call the script with parameters specifying the number of instances to create, starting IP address for the first instance, type of Linux and root password, eg. *./vboxCreateInstances.sh 5 192.168.56.20 [debian|ubuntu] PASSWORD*. The host-only network that VirtualBox creates uses typically 192.168.56.0/24, please change the starting IP address if your installation is differently configured and/or you prefer other IPs for the last quadruple. The steps, the script processes, are as follows:

1. clone the templates base image or its snapshot
2. create and start the instance (copy-on-write snapshot, two network interfaces)
3. copy some scripts and templates to each instance
4. define a static IP for the first network interface, using your start IP address parameter and incrementing on the last quadruple for each instance, for example, 192.168.56.20/24, 192.168.56.21/24, ...
5. configure the second network interface to use DHCP for external Internet access
6. set a unique hostname, postfixed with the static IP addresses last quadruple, for example, cluster20, cluster21, ...
7. modify */etc/hosts* in accordance to the changed hostname
8. add hostname/IP from the developer's workstation and all other instances to hostsfile (to not need a DNS server running on the developers workstation)
9. generate an RSA-keypair and add the developers public RSA-key to the instance's *~/.ssh/authorized_keys*
10. write an mpi.hostfile for the generated cluster
11. calling *configureAccess.sh* to exchange keys between the cluster instances and the developer's workstation

Our script requires some entries in the *config.cfg* file. Since we need to map the virtual network interfaces to the correct virtual interface on the host, we require the variable *HOSTONLY_ADAPTER_VBOX* to be set. Also, the base folder for the virtual machines is required (*BASE_FOLDER*). These settings will be configured by *setup/setupDistributedGPGPU.sh*.
The last variable we need is the name of the root account in the template (*USERNAME*, default is 'mpi').

Optional: For the base image, you might specify a snapshot instead of a virtual disk file. We did the operating systems base installation only once and continued working with a snapshot. Every time we wanted to change our template, we only needed to run our install/configuration script and additionally install important updates without much effort.

If such a snapshot is referenced as variable *TEMPLATE_ VDI_ [DEBIAN|UBUNTU]*, the script will convert it to be the cluster's base image.

The main developing environment we were using has been Windows 10 with WSL[2]. Therefore, we installed VirtualBox on Windows but executed all programs and also scripts on the Ubuntu bash in WSL. Additionally, we used a native Debian Linux as a second development environment. All our scripts work in both environments, provided that the *config.cfg* has been modified accordingly. For WSL users, keep in mind that obviously paths, but also less obviously some names between your Linux shell and Windows environment will differ. For example, the name of the host-only network interface on our developer workstation was named 'eth1' on WSL, and 'VirtualBox Host-Only Ethernet Adapter' on Windows. Virtual Box requires the Windows naming, if it is installed on Windows.

Of course, the virtual instances in your test environment won't be very powerful. But with the usage of the processor's OpenCL capabilities, an instance can work very well to test your code, and even large clusters can be created and started in a few minutes. We assigned 512MB of RAM and one CPU core per instance and typically had 5-10 instances running while we worked on the thesis. We did not observe any remarkable performance impacts on our host, which was a not above average equipped PC.

[2]https://msdn.microsoft.com/en-us/commandline/wsl/about

5 Algorithm Number 1 - Naïve Matrix Multiplication

For the first algorithm, we have chosen a straight forward naïve matrix multiplication, the typical three-nested loop everyone knows from linear algebra, implemented as OpenCL kernel. We compute $A * B = C$. The code and algorithm are pretty simple, we have decided to use it so that we can focus on the implementation using our library, not struggling with a complicated algorithm. Find the main code in *code/distributor/sampleMMul-simple.cpp* or *code/distributor/sampleMMul.cpp*. Both files contain the same example, the difference is just the number of features from the library that are used, so the simple version is clearer and easier to read.

The OpenCL host code is in the file *mmul.tpp* and the GPU kernel in *mmul.cl*. For OpenCL, the GPU kernel code is usually read from a separate text file and compiled for the corresponding OpenCL device during runtime. We have defined the kernel as a string and include the source file with the code during compile time. Because of that, we do not have a second kernel code file that we have to distribute together with our main binary to the workers. Some problems with the built-in functions from OpenMPI have motivated this decision, as explained in subsection 2.5.1.

5.1 Prerequisites

Before we start explaining our distributed matrix multiplication step by step, we want to explain how the code can be executed. We need *g++* and *mpic++* from the OpenMPI package. Start the program by executing *make run* or *make runSilent*, where the latter provides the output in order and grouped by workers/master in the files *sampleMMul.out.master* and *sampleMMul.err.master*.

For execution, you need a cluster as described in chapter 3 or 4. Alternatively, you can create an *mpi.hostfile* that includes your workstation and a number of slots for the workers. For example, *foo slots=3* would run the program on your workstation with hostname *foo* and launch two workers. Remember: one additional slot is used by the master. Verify that you can access your workstation from itself using *ssh foo* passwordless. If this does not work, check if you have generated an RSA-keypair, that your *~/.ssh/authorized_keys* contains your public key *~/.ssh/id_rsa.pub* and *~/.ssh/config* contains no faulty entries that match your workstation.

Also, you need an OpenCL driver installed. To test it, you can use our simple OpenCL test program *code/showDevices*. You need to see at least one device. By default, we use for the first worker on an instance the device 0, for the second worker the device 1 and so on. For testing and development, you might need more control or force all workers to use the same device, for example, if you are using the VirtualBox instances. They have only one CPU device but you may want to launch more than one worker per instance. In our Makefile, we have a parameter *DEBUG_FORCE_ALWAYS_USING_ACC_DEVICE_0*, which instructs the workers to use the first device, regardless of their MPI-rank. Additionally, there is a parameter *ACC_DEVICE_OFFSET* you can set in the Makefile. Imagine you have two OpenCL devices: a CPU and a GPU, and want the workers to use the GPU only. If the CPU is device 0 and the GPU is device 1, you can "skip" the CPU by setting *ACC_DEVICE_OFFSET=1*. If you want to launch more than one worker and all should share the single GPU, you can additionally activate the previously mentioned *DEBUG_FORCE_ALWAYS_USING_ACC_DEVICE_0*, the offset is added later and in the end, all workers will use device 1.
If you have more complicated requirements, you need to provide code for choosing the correct device on your own. If not, *Distributor::assignedGPU_Device()* returns the device id for your worker as we have explained in this paragraph.

5.2 Main Function

We continue describing the main function of file *code/distributor/sampleMMul.cpp* from upside down. You should already be familiar with the concept and naming as introduced in chapter 1 and chapter 2.

During development, we have used many different OpenMPI versions from v1.6 to v1.10.2. The helper function *bool getMPI_StandardVersion(int minMajorVersion, int minMinorVersion, string &output)* checks that we use at least v1.6 and returns *output* for logging.

Next, we create four Nodes: *initMatrix*, *distributeB*, *compute* and *verify*. The kernel for initializing the matrices is needed on the master only. For simplicity, our $B-matrix$ is the identity matrix so that we can compare A with C to verify the results in the verify kernel. Both kernels are very simple and need no further description. The other three kernels are more interesting and will be described in the sections 5.3, 5.4 and 5.5. Pay attention that we assigned the same kernels, different kernels or nullptr instead of a kernel, just how we needed it for each Node.
Then we add dependencies. Before we can distribute B, the master needs to initialize the matrices first. For our workers, the kernel for the Node *initMatrix* is a nullptr. The library needs to compute this Node first, because of the dependencies, but computing it means doing nothing since the kernel points to nothing. Our workers will immediately continue with the distributeB-Node, blocking until the master has computed *initMatrix* and is ready to send matrix B.

The last Node we can compute is *verify*, therefore it will become the target of our graph. Calling *Distributor d(argc, argv, verify)* additionally checks for cycles in our graph. If we have one, it throws an exception, we can log the error and quit.
Besides that, we initialize the cluster. For the master, this means we actually create the cluster or use an existing *mpi.hostfile*, distribute our binary to all instances, launch the worker-processes, configure the MPI environment and read a checkpoint if we continue computing with a restarted cluster. The workers need this function call do, but only do the two last mentioned steps.

Now, we create our Data-objects and the corresponding raw data. We need to allocate space for three matrices, the dimension $N * N$ has already been set by a command line argument.
Instead of a normal type, we are using the macro $DATA\_TYPE$. It can be modified in the Makefile, defaults to *int* and allows us to switch between *int*, $float$ and *double* for testing purposes. We use it in many of our programs, but it is not mandatory for using the library.
The Data-objects for our matrices requires three arguments for the constructor: a pointer to the raw data, which will be stored as a void* internally in the object. Next a vector of dimensions. We compute square matrices with dimension $N * N$, so our vector is $\{N, N\}$. Last argument is the size of one raw-data-element, which is $sizeof(DATA\_TYPE)$, for example 4 if we use *int*.
Our workers only need the Data-objects $B\_$ and $N\_$, but in various Nodes, so we add them as arguments to the Distributor-object. All kernels have access to their corresponding Node and the Distributor-object. The code for it is *d.addArguments({ { "B", &B_ }, { "N", &N_ }})*. The master needs additionally $A\_$ and $C\_$.

d.isMaster() and *d.getSize()* are useful aids for various programs, and *d.whoAmI()* returns the role (master or worker), hostname and MPI-rank for logging purposes. Using *d.addOutput(string)* does not directly write to the console, but stores the text in the Distributor. The same function exists for Node-objects. All this output from the Distributor and also all Nodes can be retrieved by calling *d.getOutput()*, as we do automatically at the end of the graph computation.

Finally, we can start the computations by calling *d.run()*. After that, we differentiate if all computations really finished or if we are restarting, and depending on that we log some more states.
Once the computations are finished, the workers send their output to the master, and the master additionally takes care if instances will be shut down and prepare itself for restarting if required. Speaking of shutting down, by default, shutting down the instances is deactivated. It can be activated by calling the libraries member function *setShutdownFlag(true)*, as we have explained in subsection 2.1.4.

Helper function *void writeCSV(string csvFile, vector<pair<string, unsigned>> args, vector<pair<string, unsigned long long>> results)* writes the timing data to a file, named *sampleMMul.0.hostname.csv* in our case, since we use for generating the filename a function that constructs it from the program's name, rank and hostname. The second argument is a vector of string/unsigned, were we put in everything we are interested in, concerning the current execution. Number of GPUs per instance, current cluster size[1], and our matrix dimension. The second vector contains all timing details from the Data-objects. Every send/receive/broadcast/gather/... is measured and summed up separately. We log here only the timings of the current process, the master does not collect and combine this from the workers. We wanted this information for finding bottlenecks and think it is of more use if we have this data separated per process.

Last step: clean-up. Nothing special for the library here. Just keep in mind that the Data-Objects do not create or copy data, they only use pointers. So we actually delete the raw data from the heap by calling *delete[] static_cast<DATA_TYPE*>(B_.get())*. This line of code is equivalent with *delete[] B*, which would be shorter, but less helpful for us explaining many different features and use cases of the library.

5.3 Master and Worker Kernel - Distribute B-matrix

The distributeB-Nodes kernel *kernel_DistributeB(Node &n, Distributor &d)* contains of exactly one line of code: *d.getArgument("B")->bcast_M_to_W()*. We dedicate it its own section only because we want to show an example where master and worker use both the same kernel function and it describes the interface of a kernel in a simple way. Broadcasting with raw MPI functions also executes exactly the same code with the same data as arguments for sender and receivers, our library does nothing surprisingly here. But it simplifies. No need to think about the ID of the target node or if the intra- or intercommunicator is required. The number of elements is stored in the Data-object, and although we did not store the original type of the raw-data, we know how many bytes one element has and so can transfer the complete B-matrix from master to all workers. Our Data-Object has been stored as an argument in the Distributor, which is one of the kernels two parameters. If we would have used it as an argument for the Node, we would have needed to add it to other Nodes like the compute-Node too.
OK, the kernel has not exactly one line of code, we also need to return an error code and did not combine these two lines. Each kernel needs to report an error code. The library does not stop executing subsequent nodes if one has reported something non-zero, but we bitwise or the return codes of all nodes and return it after calling *d.run()*.

[1] If we have restarted, the final, reduced cluster size will be logged, which is uninformative. But since we use this function for benchmarking and there we do not restart the cluster, it's fine for this program.

5.4 Masters Compute Kernel - Distribute Chunks & Combine Results

For this section, the code references to the file *code/distributor/sampleMMul.cpp*. The compute kernel is a typical example where master and worker have almost nothing in common. The master sends chunks that the workers receive, and for the results, communication works the other way round. Everything else is completely different.

Remember that we have already described how the library manages idle/busy workers and restarting the cluster in the subsections 2.1.3 and 2.3.1. They have referred to this kernel *kernel_ComputeOnMaster()* and the corresponding function *waitForWorker()*. We continue now with the few previously unmentioned steps.

Our master has to divide the work (=A-matrix) into chunks. The size of one chunk is defined by $ROWS$, we calculate it by choosing the minimum of the quotient of the dimension divided by the number of workers and upper bound *MAX_ROWS_PER_WORKER*. Convenient is splitting the data: *d.getArgument("A")->sliceSize(ROWS*N)* takes the Distributors argument "A" and cuts it into pieces of size $ROWS * N$. The last piece might be smaller if $ROWS$ divides $N * N$ with remainder. The resulting vector *aChunks* contains only pointers to the start of the current chunk, nothing has been copied. This operation is fast. A similar function *Data::sliceParts()* exists if you want to split the data into for example as many parts as you have workers.

After all chunks have been distributed, the master needs to wait for the workers to collect the last results. The Distributor has functions for the number of currently idle workers and the total number, we can loop until all have finished their work and therefore both functions return the same number: *while (d.availableWorkers() != (d.getSize()))*. The loop's body calls the function that waits for an arbitrary workers result and merges it into the C-matrix.

We do not have to free any data here. Only in function *waitForWorkers()* we temporarily need additional space, because we do not know in advance which worker will send us the next results until we have received the data. So we do not know where in C-matrix we could point. Our temporary buffer is again a Data-object. We size as we did for creating the chunks. If we receive the last chunk, which might be smaller, the Data-object automatically sets its internals to the received number of bytes. We lose the number of dimensions during slicing, but this is no problem. And of course, we could reconstruct the dimensions since we know that one column is N wide.

5.5 Workers Compute Kernel - Compute Chunks

In the *kernel_ComputeOnWorkers()* our workers create temporary buffers for the expected *aChunk* and result *cBuffer*. Again we use a Data-object for convenient receiving of data. Since the worker does not know how many chunks to expect, we loop as long as we receive work. Once the worker has nothing more to compute, the master sends one last Data-object. This time it does not contain valid raw-data, but has a special tag: *Data::TAGS::TERMINATE_TAG* or *Data::TAGS::RESTART_TAG*. The shutdown behavior has been discussed in subsection 2.3.1.

The function for the computation is in *mmul.tpp*. We will not explain it since it has no influence on our library and does not use anything of it except the function that assigns OpenCL devices to workers. In case you are unfamiliar with OpenCL, it is a good example to see how OpenCL works since it is quite simple. Just pay attention that the structs *cl* and *cl_param* as well as all functions prefixed with *clu* are helpers and do not belong to the OpenCL standard. The GPU kernel is typically a separate file, here *mmul.cl*, but as mentioned at the beginning of this chapter we have wrapped the code in a string instead of using it as plain text.

5.6 Benchmark Results

Our sample algorithm has been chosen because it is known that it achieves better performance when executed in parallel, although our naïve algorithm is far from being an efficient one. The benchmark results in table 5.1 and the charts 5.1 and 5.2 show that our library and the distribution work as expected.
We have used different matrix sizes $N = \{500; 1,000; 2,000; 3,000\}$ per dimension, distributed up to 24 chunks and overloaded our instances so that two workers on one instance shared the single GPU a g2.2xlarge instance on Amazon EC2 provides.

cluster size	time (ms)
1	2,476.00
2	1,923.75
4	1,231.50
6	993.75
8	849.25
10	814.00

Table 5.1: Computation Time (ms)

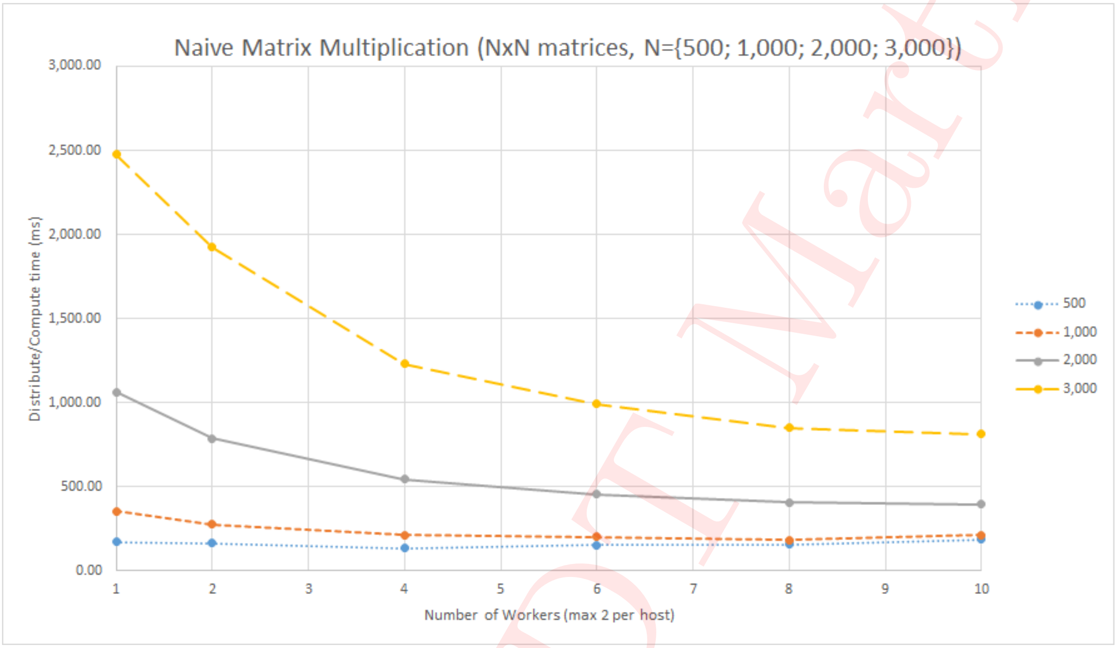


Figure 5.1: Performance for distributed naïve matrix multiplication

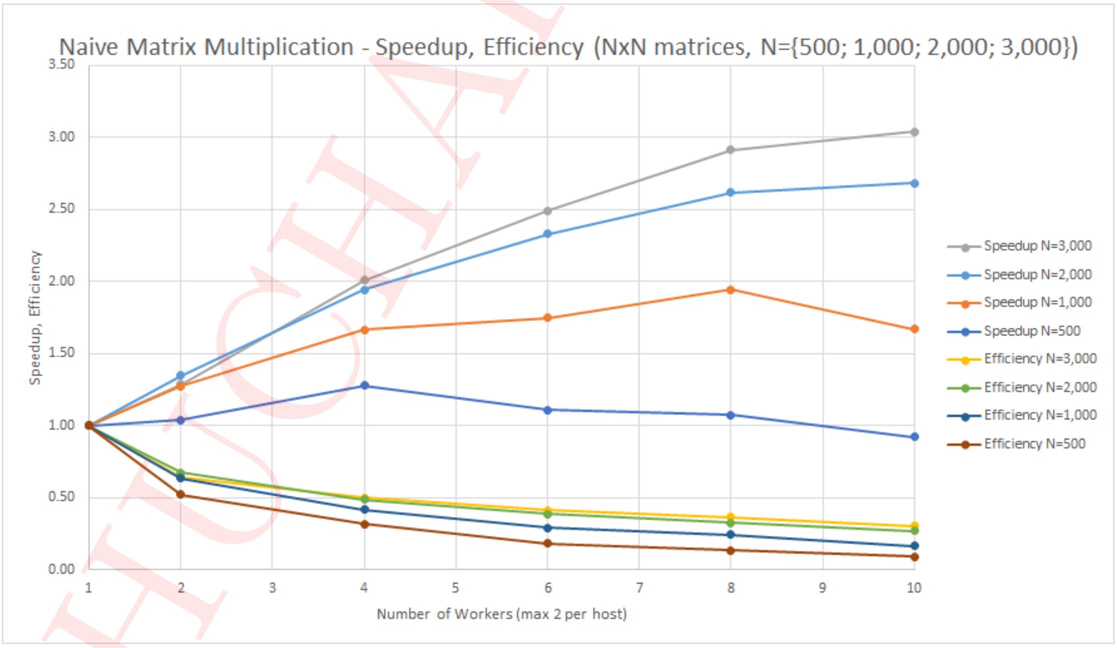


Figure 5.2: Speedup/efficiency for distributed naïve matrix multiplication

5.7 Conclusion

This chapter was written to demonstrate the usage of our library. Still, we would like to call attention to the level of simplification the library has brought to enable your program being executed on a GPU cluster instead of a single machine. Of course, hooks like reassembling results in the correct order asynchronously require additional work, and compared with the naive single version of our sample these changes result in the largest number of additional lines of code.

But then, executing the prepared algorithm on a (GPU) cluster is a piece of cake. Assign nodes, define dependencies, compute. All communications using MPI are trivial one-liners. The creation of the cluster comes for free, with no need of controlling anything, and on an arbitrary number of instances.

Simplification was one of our main goals, we believe that our library fulfills that.

6 Algorithm Number 2 - SpMV GMRES Solver

The second algorithm we have implemented is a sparse matrix-vector multiplication, using the generalized minimal residual method (GMRES). It is an iterative method to solve linear equations by approximating the solution, and can so solve systems with a large number of variables in a few iterations. For more profound explanation please refer to Saad [8] or a summary like[1].

The implementation has been provided by the courtesy of Mr. Kawamura (JAIST) and Fujitsu, our work was limited to do the necessary steps to split the data, distribute the chunks to the instances and combine their subresults. They have developed several different methods to compress a sparse matrix and compute the matrix-vector product. We have implemented only one method, CSR[2]. Since the base implementation is a trade secret of Fujitsu, we will not give more details on the code nor have it included in our repository [9].

6.1 Necessary Changes - Splitting the Input Data

For the computation, the kernel needs the input matrix in a compressed storage format (CSR). As described in equation 6.1, the information of source sparse matrix A will be rewritten to three different arrays:

1. val, which contains all non-zero values in order of their position in A
2. col_Index, which enumerates the row index of each nonzero element of A
3. and row_Ptr, which contains the number of non-zero of the current row plus all previous rows.

The blue and green numbers in subscript in equation 6.1 describe the corresponding row respective column index for each non-zero element in all three arrays and the matrix.

For the distribution, each worker should now receive one part of the CSR-object. Each chunk needs to contain all elements of a number of rows of source matrix A. For example, the three arrays need to be split as described by the red and blue boxes in equation 6.1

[1]https://en.wikipedia.org/wiki/Generalized_minimal_residual_method
[2] https://de.wikipedia.org/wiki/Compressed_Row_Storage

if we want to split the data into two chunks. For the first two arrays, this works straight forward, *row_Ptr* needs to have its counter reset to zero for each block we generate.

$$
\begin{aligned}
A &= \begin{pmatrix}
\underset{(0,0)}{10} & 0 & 0 & \underset{(0,3)}{11} & 0 \\
0 & 0 & \underset{(1,2)}{12} & 0 & \underset{(1,4)}{13} \\
\underset{(2,0)}{14} & \underset{(2,1)}{15} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \underset{(3,4)}{16}
\end{pmatrix} \\
val &= \begin{pmatrix} \underset{(0,0)}{10} & \underset{(0,3)}{11} & \underset{(1,2)}{12} & \underset{(1,4)}{13} & \underset{(2,0)}{14} & \underset{(2,1)}{15} & \underset{(3,4)}{16} \end{pmatrix} \\
col\_Index &= \begin{pmatrix} \underset{(0}{0} & \underset{)}{3} & \underset{(1}{2} & \underset{)}{4} & \underset{(2}{0} & \underset{)}{1} & \underset{(3)}{4} \end{pmatrix} \\
row\_Ptr &= \begin{pmatrix} \underset{(0)}{0} & \underset{(1)}{2} & \underset{(2/0)}{4/0} & \underset{(1)}{6/2} & \underset{(2)}{7/3} \end{pmatrix}
\end{aligned}
\tag{6.1}
$$

Now we can compute the vector for each chunk, which will contain the overall results for the indexes matching the rows we distributed from the source matrix A. Figure 6.1 shows the graph for this problem, imagine it will be split and distributed to four workers.

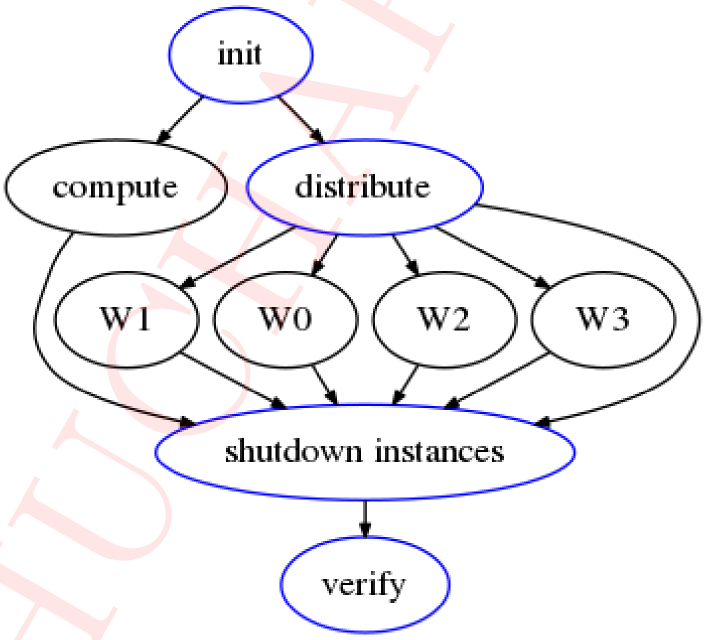


Figure 6.1: Graph for a distributed SpMV GMRES solver

6.2 Necessary Changes - Adjusting the Number of Rows

The original algorithm worked with square matrices. There was no need to distinguish between rows and columns. The distributed version now operates on chunks, the number of rows will be less than the total number of rows and unequal to the number of columns. This has falsified several internal arrays. We have detected and corrected some flaws, and after a reasonable time spent on investigating on errors, we believe that the false row counter is responsible for various false loop counters and uninitialized data on temporary arrays the algorithm uses. But we decided to not spend more time on finding and correcting the errors:

- For the thesis, we should distribute using an existing algorithm. We should not spend time on writing new algorithms or modifying an algorithm too much.
- First checks have stated that we cannot expect a remarkable better performance from the parallelized version. For details, see section 6.3.
- More efficient algorithms than CSR would require much more time to be corrected and can be expected to perform worse because of additional communication required between worker kernels.

Because of that, we decided to not correct the algorithm. The distribution works basically but is not efficient for this algorithm/problem size. The original algorithm has not been designed to work on rectangular matrices, so it can't compute correct results. Spending manpower on implementing an algorithm that will be outrun by already existing solutions would not be of any use for all parties involved.

6.3 Benchmark Results

Since the results of the worker kernels are erroneous, we know that our benchmarks are likely to deliver wrong timing results. As explained in section 6.2, we stopped working on a modified version of the algorithm used for the implementation.

Table 6.1 and figure 6.2 show the execution time for the program, starting with the modification of the source matrix to create chunks and stopping after all computed results have been combined.
The computation for cluster size 1 executes our code, but since it splits the CSR-object into one chunk and distributes it to one worker, it does basically the same except for additional overhead for transferring the data from master to worker and back. The source sparse matrix we used for benchmarking contains 81,736 elements and 3,562 rows/-columns. The time spent on transferring the corresponding CSR-object is 9ms, 5.96% on the total computation time.
The drastic decrease in performance for the execution on 2 or more workers is an interesting result. Network transfer time increases to 19ms, as the chunks need to be sent sequentially from the master to each worker. But the main loss happens on executing the

worker kernel, unexpected for chunks with half the size of the whole data that was used for the 1-worker cluster. In theory, computing on fewer elements should be faster. We believe that the additional time spent on the computations is related to the many NAN the internal arrays contain because of erroneously false sized arrays and uninitialized data. This symptom has been reported by others and is matching our observations[3][4][5].

cluster size	total time (ms)	network time (ms)
1	151.00	9.00
2	231.75	20.00
3	222.75	15.00
4	213.25	17.00
5	239.50	59.00

Table 6.1: Computation Time (ms)

Still, even if we would achieve a better performance and could, for example, achieve an unrealistic, perfect linear speedup, our total computation time for two workers would be approximately $(151ms - 9ms)/2 + 20ms = 91ms$, which would be a speedup of $91ms/151ms = 0.60$, and an efficiency of $0.60/2 = 0.30$. This result might be OK for the parallel implementation of CSR, but Mr. Kawamura and Fujitsu have developed more sophisticated algorithms that already outrun our speedup.
Parallelizing this faster methods would require synchronization and combining temporary results in every iteration of the main GMRES loop, which are about 200. We expect that for such short computation time in the range of 150ms this number of additional communication between all workers will ruin every possible speedup that might be achieved by the smaller data chunks.
For completeness, the original code had a computation time of 161.40ms. The overhead for the launch of the MPI processes has not been counted, these timings refer to the plain computation time only. The time required for generating the chunks is negligible, we believe the better performance of our algorithm is achieved because of the slightly restructured code, which is now grouped in smaller functions and so will probably give the compiler more possibilities for optimizations.

Summarizing, we were able to split, distribute and recombine the data for the SpMV GMRES solver. The wrong results base on the different prerequisites that came with the generated chunks. Small modifications are normal when parallelizing an algorithm, but we believe we would need to rework too much. However, achieving a good speedup and efficiency is difficult if the total execution time is as small as for our problem size.

[3]https://connect.microsoft.com/VisualStudio/feedback/details/498934/big-performance-penalty-for-checking-for-nans-or-infinity
[4]https://stackoverflow.com/questions/24929163/pownan-is-very-slow
[5]https://randomascii.wordpress.com/2012/05/20/thats-not-normalthe-performance-of-odd-floats/

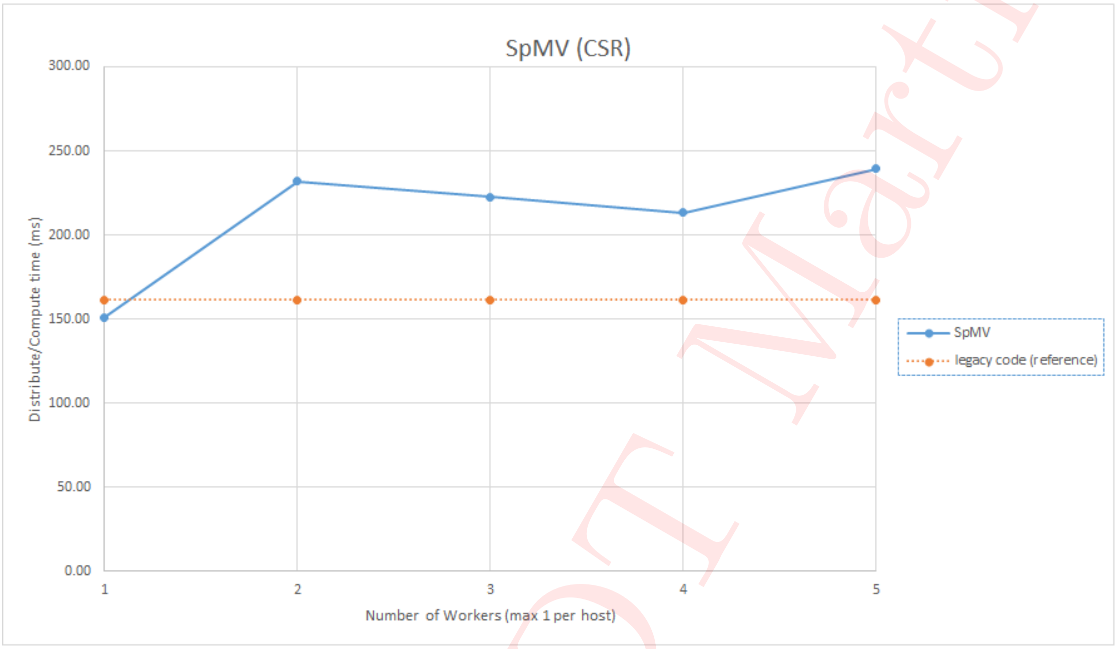


Figure 6.2: Performance loss for distributed SpMV GMRES Solver

The necessary work to distribute the workload on a GPU cluster was again straight forward. Once the code has been prepared to represent a workflow, the tasks like communication, distribution, and computation were negligible.

7 Conclusion

7.1 Summary

This thesis evaluated if GPU instances on cloud providers can be used for GPGPU clusters in a simple way. We have automated the creation and configuration of Amazon EC2 instances. Concerning security and network restrictions, we implemented different possibilities how to design the cluster, using an on-site master instance or having all required instances running in the cloud. Launching a cluster is fast, and depending on the number of hours it is used per day/month/year it might be a cost effective alternative instead of maintaining a private GPGPU cluster on-site.
Using a small benchmark suite, we checked the cloud instances performance and compared it with standalone hardware.

Cloud instances as workhorses have been our main target, but we also implemented a cluster for testing and development purposes that can be operated on even weak hardware. We used VirtualBox and developed scripts to configure and launch a cluster. As guest operating systems we have used Ubuntu and Debian.

The library we have written simplifies typical tasks that are required for many distributed systems. It aims at a cost effective use of cloud instances, giving the possibility to shut down unused instances. Although we offer the programmer to use MPI without restrictions, we have written wrappers and functions for several tasks that require much less care on MPI's internal states and provide a more convenient usage.

We differ from typical schedulers used on on-site clusters, where a program will automatically be launched on that instances, that the scheduler has assigned to the program. We implemented the core features, distribution of the binary and its launch in the library, so that we do not need such a scheduler being installed and configured on our cloud clusters. For people with experience on MPI clusters this might be confusing, but for our bare installed and most of the time short running cloud instances, this uncommon concept makes more sense.

Some frameworks like condor[1] work with an interface that focuses on send/recv/exec per node. In theory, we do this too. But since our Data-objects make send/receive tasks always a one-liner, our code would not have become easier. So, we have only our master/worker kernels and do all work within them.

[1]http://research.cs.wisc.edu/htcondor/

Also, send/receive functions from condor are done as prerequisites for executing a program (which they call node) on the assigned worker. This might be compared with our distribution of the binary at the beginning of the execution and transmit of Data-chunks after each workers compute unit. Since we use the library to create and terminate a cluster, there would be nothing before and after the library that could receive or send data.

As a prove of usability, we implemented two algorithms: a naïve matrix multiplication using OpenCL for the GPU kernels, and a sparse matrix-vector multiplication using CUDA for the GPU kernels. With our library, both algorithms could be implemented as a distributed system in a convenient way.

7.2 Future Work

We have used Amazon EC2 for our implementation and tests since they were first who offered GPU instances in the cloud. In the meanwhile, other companies have added offers for cloud GPGPU. We assume that adopting our method to run on other clouds will be an easy, and perhaps interesting improvement.

Recently, Amazon has made images of their Linux distributions available for on-site tests with virtualization software including Virtual Box. Adding such an image additional to our Debian/Ubuntu test images would be useful.

The feature to restart and shrink the cluster has initially been implemented to save costs. Now, since EC2 has reduced the payment time unit from one hour to one minute, this feature does not save money any more. But it can be used to make snapshots, where computations can continue in case the cluster crashes. MPI is quite fragile on distributed networks, a short interruption can cause the process to crash. For long running computations, snapshots might be very useful to save time and costs.

Only a few of the many data transfer features that MPI offers have been implemented as simplified wrappers. Using the provided code as a template, adding more features will not be difficult. By using the library for more algorithms that should compute in the cloud, some more of these MPI features will probably be required and the library can be adopted on demand.

Our work is open source, we hope that it is helpful, your comments are welcome and we invite everyone to help develop it further.

Bibliography

[1] Amazon. Amazon EC2 Instance Types, Jul 2017.

[2] Amazon. *Amazon Elastic Compute Cloud User Guide for Linux Instances*, Jul 2017.

[3] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*, Nov 2015.

[4] Ferry Boender. Multiple VirtualBox VMs using one base image (copy-on-write), Sep 2011.

[5] NVIDIA. TESLA K20 GPU Accelerator, Nov 2012.

[6] NVIDIA. TESLA K520 GPU Accelerator, May 2012.

[7] NVIDIA. Tesla Kepler GPU Accelerators, Oct 2012.

[8] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[9] Tomoki Kawamura, Yoneda Kazunori, Takashi Yamazaki, Takashi Iwamura, Masahiro Watanabe and Yasushi Inoguchi. A compression method for storage formats of sparse matrix in solving the large scale linear systems. Advances in Parallel and Distributed Computational Models (APDCM), Mar 2017.