NETWORK-AWARE DATA PREFETCHING OPTIMIZATION OF COMPUTATIONS IN A HETEROGENEOUS HPC FRAMEWORK

Paweł Rościszewski[1]

[1]Department of Computer Architecture,
Faculty of Electronics, Telecommunications and Informatics,
Gdańsk University of Technology, Gdańsk, Poland

ABSTRACT

Rapid development of diverse computer architectures and hardware accelerators caused that designing parallel systems faces new problems resulting from their heterogeneity. Our implementation of a parallel system called KernelHive allows to efficiently run applications in a heterogeneous environment consisting of multiple collections of nodes with different types of computing devices. The execution engine of the system is open for optimizer implementations, focusing on various criteria. In this paper, we propose a new optimizer for KernelHive, that utilizes distributed databases and performs data prefetching to optimize the execution time of applications, which process large input data. Employing a versatile data management scheme, which allows combining various distributed data providers, we propose using NoSQL databases for our purposes. We support our solution with results of experiments with real executions of our OpenCL implementation of a regular expression matching application in various hardware configurations. Additionally, we propose a network-aware scheduling scheme for selecting hardware for the proposed optimizer and present simulations that demonstrate its advantages.

KEYWORDS

Parallel Computing, High Performance Computing, Heterogeneous Environments, OpenCL

1. INTRODUCTION

The market of electronic hardware is developing in extreme pace, making sophisticated computing devices accessible to households. Research and development departments of hardware manufacturing companies compete in designing new architectures and accelerators. HPC (High Performance Computing) systems no longer can be considered as sets of very expensive devices forming a cluster, physically installed in one room. The HPC field has to deal with increasing heterogeneity of the systems and it should be taken into account that the parallelization is performed on many levels. We should be able to combine concepts as Grid Computing [1], GPGPU [2] and Volunteer Computing [3] into one multi–level parallel design.

Our parallel processing framework KernelHive [13] is able to perform parallel computations on a set of distributed clusters containing nodes with different types of computing devices. We presented the KernelHive system and its performance capabilities in [4] and proposed an execution optimizer focusing on energy efficiency in [5]. This paper is an extended version of [14], where we added data intensity capabilities to the KernelHive system. For this purpose we proposed MongoDB [6] database as a backend. For our experiments, we used our solution to the regular expression matching problem [7]. This allowed to propose a new data prefetching optimizer, which we extend by proposing a network-aware internal scheduler.

DOI : 10.5121/ijcnc.2014.6506

The outline of this paper is as follows: in Section 2 we formulate the addressed problems by introducing data intensity in HPC systems in 2.1, describing the existing KernelHive architecture in 2.2 and justifying the need for network-aware scheduling in 2.3. We present our solutions in Section 3 including data addressing in 3.1, selecting database system in 3.2, the new prefetching optimizer in 3.3 and network-aware scheduling scheme in 3.4. The description and results of our experiments are presented in Section 4 with real parallel executions on a single device in 4.1, in a heterogeneous environment in 4.2 and simulations of executions with scheduler comparison in 4.3. Finally, we conclude our work in Section 5.

2. PROBLEM FORMULATION

2.1. Data Intensity in HPC Systems

From the parallelization point of view, the spectrum of computational problems in general can be structured as shown in Figure 1. The parallelization process requires dividing the problem into subproblems, solving them independently by parallel processes and finally merging the results. Certain problems require only partitioning the input data into chunks, which are processed indepentently. Problems of this type are called embarassingly parallel and in Figure 1 are located in the compute intensive corner. Until this work, the KernelHive system was dealing only with this type of problems (e.g. breaking MD5 hashes).

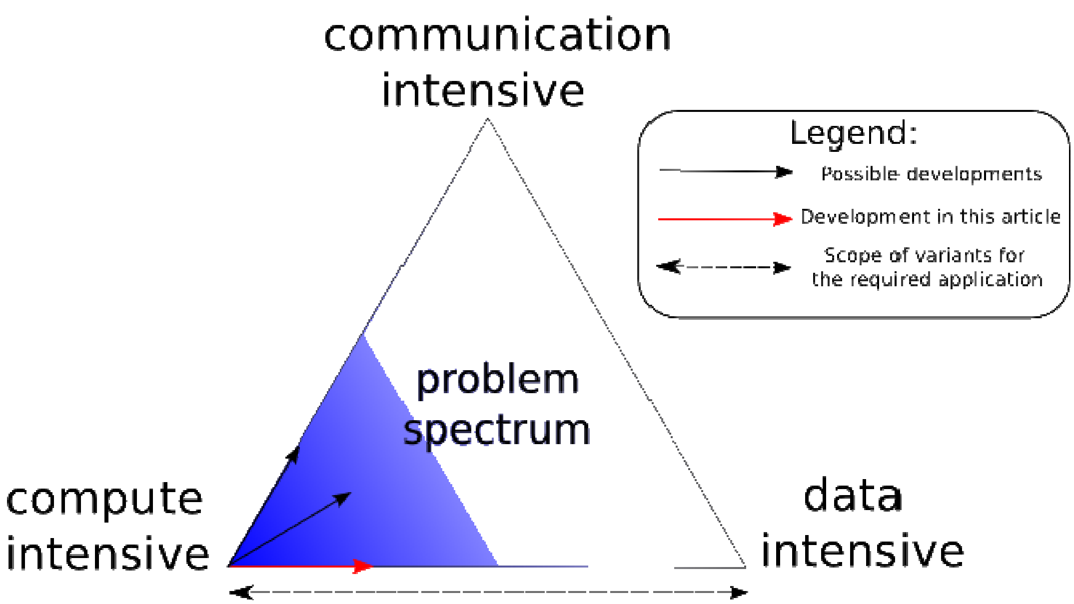

Figure 1. Spectrum of Computational Problems from the Parallelization Viewpoint

The black arrows on Figure 1 show possible directions of development of the KernelHive system. Moving in the direction towards communication intensive vertex of the problem spectrum, we would be dealing with applications that require communication between the processes (for example for frequent updating intermediate results). This direction of development should be addressed in the future. The dashed arrow at the bottom reflects profiles required from an application for our experiments. We use a regular expression matching application with configurable data intensity as proposed in [18].

2.2. Overview of the Existing Architecture

The system architecture so far is shown in Figure 2. Using the graphical interface, the user defines an application in a form of a directed acyclic graph. Graph nodes correspond to computational tasks and are selected from a repository of predefined node types (e.g. processor, partitioner, merger). Each node is provided with a number of computational codes corresponding to its role. The edges of the graph denote the direction of data flow between the tasks.

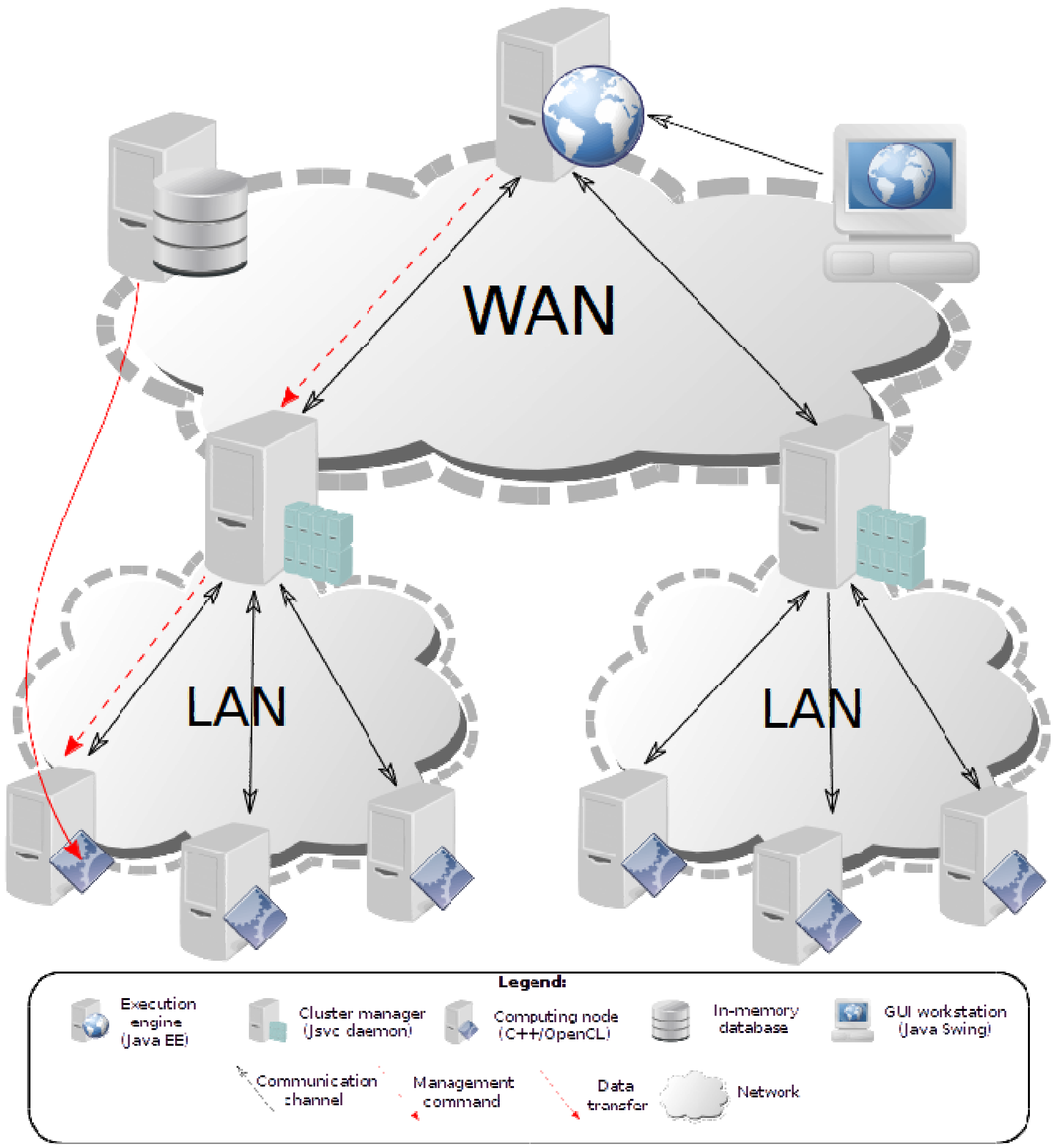

Figure 2. Basic Architecture of the Parallel System

Applications for the system can be defined using our graphical tool called *hive-gui* (Java Swing application), however they are represented in a XML format, allowing other front-ends to use the parallel system. A tested example of such front-end is the Galaxy Simulator [9] which was extended by a plugin for KernelHive executions. The application XML, along with data addresses are dispatched for execution by a SOAP web service.

Analysis and deployment of the applications is performed by the *Engine*, which is a high-level Java EE application. All of the subject modules report their state to the engine, keeping a live representation of the whole system in the engine. Thus we can define rules of scheduling the tasks that have a rich view of the available infrastructure and its state.

One of the distinct features of KernelHive is that it is designed to combine multiple distributed clusters. The only requirement towards a cluster is that there should be one machine playing a role of an entry point to the cluster, which has to be visible (in terms of network) by the *Engine*. To address this requirement, the system utilizes the *Cluster* subsystem, working as a Java system daemon. It is a middleman between the engine and computing devices, which are managed by C++ daemons, capable of dynamic compiling and running OpenCL [10] computational code.

2.3. Need for a Network-aware Scheduling Scheme

The results of the experiments with prefetching optimizer presented in [14] revealed, that in case of data packages of similar size and efficient data management, network is constantly loaded. The benefits of prefetching in a more complex environment (Figure 8) are significantly smaller than in case of a single device (Figure 6). This means, that the more computational machines in one cluster, the less would be the benefit of data prefetching. What is more, even without prefetching mechanism, downloading the input packages to machines from the same cluster would cause overlapping of communications and hence longer execution time. The internal optimizer for scheduling (baseOptimizer in Figure 4) used for the experiments applied a round-robin scheme: the machines were selected for computations in a random order. This is the place, where we should propose a more sophisticated, for example network-aware approach.
The ideas of network-aware scheduling are present in the field of heterogeneous HPC systems. For example, the authors of [15] propose a framework for heterogeneous HPC systems, which allows to calculate schedules based on network performance information during application runtime. This approach is claimed to improve the performance by a factor of 5 comparing to homogeneous scheduling schemes. Similar approach was used in [16] to satisfy deadlines on real-time tandem task graphs. Based on these ideas, we should be able to introduce a network-aware improvement to our system.

3. PROPOSED SOLUTION

3.1. Flexible Data Addressing

In the embarassingly parallel applications considered so far, the time of sending the input data to the computational node was negligible: the data could be considered as a part of the management command and stored in memory. However, in case of larger data a method of storing data on hard drive has to be employed, should it be a database system or filesystem. What is more, it should be noted that the bandwidth between the cluster manager and engine and, more importantly, data server, is significantly lower than in the local network between cluster manager and computational nodes. In case of larger input data, we propose an approach, where management commands contain only addresses of data packages. The addresses are defined in a versatile way and consist of:

- hostname – the TCP hostname of the data server
- port – the TCP port of the data server
- ID – identifier of the data package unique within the data server scope

This approach has two main advantages:

- tolerates different technologies for the data servers, which allows to adjust the data server to the characteristics of given application and deployment
- grants the possibility to move the data between the servers during the application runtime and changing the addresses in management commands at low cos

In this paper we show examples of exploring both these advantages. For the first one, we propose using MongoDB key–value store with the GridFS [11] drivers as the technology for data servers. The power of the second advantage is exposed on the example of communication and computation overlapping by prefetching input data to local servers.

3.2. NoSQL Data Servers

There are numerous technologies designed for storing and accessing big data. The concept of filesystems evolved from basic structures for storing data on local hard drives to sophisticated distributed filesystems. These solutions are closely connected with the operating system issues, like access control and hierarchical organisation of data. Because of this, they often introduce some constraints on file names, limit number of files in a directory etc. However, filesystems are widely used as the backend for HPC systems, which have to be aware of the characteristics of used filesystems.

Another important approach towards storing data is relational databases. Database management systems (DBMS) deal with the low level details of storage and hide them from the user. They provide wide functionality of storing, retrieving, filtering data, often with regard to transactions and cascading of operations. The data is modeled in a rigid form of relational tables with columns corresponding to certain object attributes and rows representing some objects.

In case of HPC systems, we rarely require the database to understand the model of our data. Often, we just need to store a big file and keep an address to refer to it later. However, we would like to benefit from the low–level internal transparency offered by the database systems. For this reason, we propose to use a NoSQL database for our reasons.

The NoSQL [12] concept is close to the relational databases, however abandons the rigid representation of data. For our experiments, we chose the most popular NoSQL database at the time, MongoDB. This database system comes with an extension called GridFS. The extension is actually a functionality of the MongoDB drivers that allows automatic dividing the data to chunks, storing them separately, but keeping information about the whole files in metadata.
Another reason for using MongoDB in our heterogeneous system is that it offers mature driver implementations for different programming platforms. We benefited from the implementations in:

- python – for the input data package generator
- C++ - for the program on the computing devices to download the input database
- Java – for the cluster manager to perform the data prefetching

3.3. Data Prefetching Optimizer

The KernelHive *Engine* defines a IOptimizer programming interface, listed in Figure 3.

Figure 3. *IOptimizer* interface

```
public interface IOptimizer {
        /**
         * Given a submitted workflow and available infrastructure,
         * return a list of scheduled jobs with assigned devices to
         * be deployed by the Engine.
         */
         public List<Job> processWorkflow(Workflow workflow, Collection<
              Cluster> infrastructure);


}
```

The input of each Optimizer implementation consists of:

- *Workflow* – class representing the whole application workflow, including individual jobs and relations between them. The optimizer has access to the state of each job (e.g. pending, ready, prefetching, prefetching finished, finished)
- collection of *Clusters* – a set of instances of *Cluster* class, each representing a collection of computational nodes. The optimizer has access to the full infrastructure model, including the computing devices, their characteristics and current state.

The value returned by the optimizer is a list of jobs, that were scheduled for execution. Additionally, the optimizer should change the states of affected jobs and devices.

The interface is general enough to allow its implementations to focus on different criteria and perform diverse tasks. It is also possible to combine several optimizers to achieve a complex goal. We have already implemented scheduling optimizers aimed for dynamic assignment of jobs that became ready for execution to available devices according to certain criteria (e.g. performance, energy efficiency).

The new *PrefetchingOptimizer* implementation requires an internal optimizer for scheduling. This way, choosing the hardware for computations can be done by an exchangeable component. Such base is extended by a data prefetching mechanism, listed in Figure 5. The optimizer implementation keeps the information about currently performed prefetchings in a map. Each prefetching process is represented by a key-value pair of jobs:

- key – a job that is being processed (data has already been downloaded by the worker)
- value – next job assigned to the same computational device as the key job, however not yet scheduled for execution – only for data downloading

Using a data structure defined this way, the tasks of the optimizer are as follows:

- if a key job has ended and the prefetching for the corresponding next job is over, mark the latter as scheduled for execution

```java
public class PrefetchingOptimizer implements IOptimizer {

        private Map<EngineJob, EngineJob> prefetchingMap = new HashMap<EngineJob, EngineJob>();
        private IOptimizer baseOptimizer;

        public PrefetchingOptimizer(IOptimizer baseOptimizer) {
                this.baseOptimizer = baseOptimizer;
        }

        @Override
        public List<Job> processWorkflow(Workflow workflow,
                        Collection<Cluster> infrastructure) {
                // First schedule jobs that were prefetched
                List<Job> scheduledJobs = schedulePrefetchedJobs();

                // Then schedule jobs to free resources
                scheduledJobs.addAll(baseOptimizer.processWorkflow(workflow, infrastructure));

                // Then start prefetchings
                List<EngineJob> processingJobs = workflow.getJobsByState(Job.JobState.PROCESSING);
                processingJobs.removeAll(prefetchingMap.keySet());

                if(processingJobs.size() > 0) {
                        List<EngineJob> readyJobs = workflow.getJobsByState(Job.JobState.READY);
                        Iterator<EngineJob> readyIterator = readyJobs.iterator();
                        for(EngineJob pj : processingJobs) {
                                if(!readyIterator.hasNext())
                                        break;

                                EngineJob prefetchingJob = readyIterator.next();

                                prefetchingJob.device = pj.device;
                                prefetchingJob.runPrefetching();

                                prefetchingMap.put(pj, prefetchingJob);
                        }
                }

                return scheduledJobs;
        }

        private List<Job> schedulePrefetchedJobs() {
                List<Job> scheduledJobs = new ArrayList<Job>();
                List<EngineJob> toRemove = new ArrayList<EngineJob>();
                for(EngineJob processingJob : prefetchingMap.keySet()) {
                        if(processingJob.state == JobState.FINISHED) {
                                EngineJob prefetchingJob = prefetchingMap.get(processingJob);
                                if(prefetchingJob.state == JobState.PREFETCHING_FINISHED) {
                                        prefetchingJob.schedule(prefetchingJob.device);
                                        scheduledJobs.add(prefetchingJob);
                                        toRemove.add(processingJob);
                                }
                                // Do not schedule new job if we are waiting for prefetching
                                else prefetchingJob.device.busy = true;
                        }
                }

                for(EngineJob tr : toRemove)
                        prefetchingMap.remove(tr);

                return scheduledJobs;
        }

}
```

Figure 4. The new *PrefetchingOptimizer*

- let the internal optimizer perform the scheduling of jobs that became ready for execution (due to the workflow dependencies) using hardware that became available (because it finished its computations or has been just connected to the system)
- ensure, that for each currently processed job, there is a corresponding job, for which the input data is being prefetched (provided there are some jobs ready for execution)

The optimizers *processWorkflow* method is called by the *Engine* upon every event that changes the aforementioned states of jobs and hardware, including finishing a job, finishing a prefetching, submitting new workflow or connecting new hardware.

After each call of this method, the list of scheduled jobs returned by the optimizer is sent by the *Engine* to appropriate *Cluster* subsystem instances. Then, the jobs are forwarded to the assigned machines, where the *Unit* subystem listens for jobs to run. Finally, the adequate *Worker* binary is executed. It downloads the necessary input data, application code, builds it and runs the computations.

When the computations are finished, the output data is saved in a previously configured database. A management command is send back through the *Cluster* to the *Engine*, containing the resulting data package ID. In case of final results, the ID is used to download them upon users request. In case of intermediate data, the ID is used by following jobs in the workflow.
Figure 5 shows the system design after introducing distributed data servers and the optimizer.

3.4. Network-aware Scheduling Scheme

The problem described in Section 2.3 can be to some extent reduced in case of a computing environments with multiple clusters (or more precisely: networks) available. The idea is to schedule the tasks equally between the clusters, in order to minimize the number of communication overlaps.

As mentioned in Section 3.3, the *IOptimizer* interface allows to create a hierarchy of optimizers, each of which could be responsible for different type of optimization. The *PrefetchingOptimizer* presented in Figure 4 uses an internal optimizer for scheduling. Instead of a round-robin scheduler utilized until this work we propose a scheme, where machines from less network-loaded clusters are selected first. Within the cluster, the scheduling can be done by another level of internal scheduler, in this case the previous round-robin one.

4. EXPERIMENTS

The proposed solution was tested in a series of experiments. We measured the execution times of a regular expression matching application with different numbers of input data packages. The data packages are 20MB files of random characters, generated and stored in MongoDB by our generator script. Additionally, each package is prefixed with a header containing the needle and haystack sizes, and the needle itself. In the experiments we searched for the occurrences of the pattern ”a*b*c*d”. The details of the application are introduced in [18].

The prefetching algorithm should enhance the systems performance provided the WAN network shown in Figures 3 and 5 brings significant delays and bandwidth limits. To reflect this situation during the experiments, the source database was hosted on a server in France, while the computations took place in our department lab in Poland.

For assesing the fitness of the proposed network-aware scheduling scheme, we used a prototype version of a large-scale HPC system simulator, which concepts were described in [17].

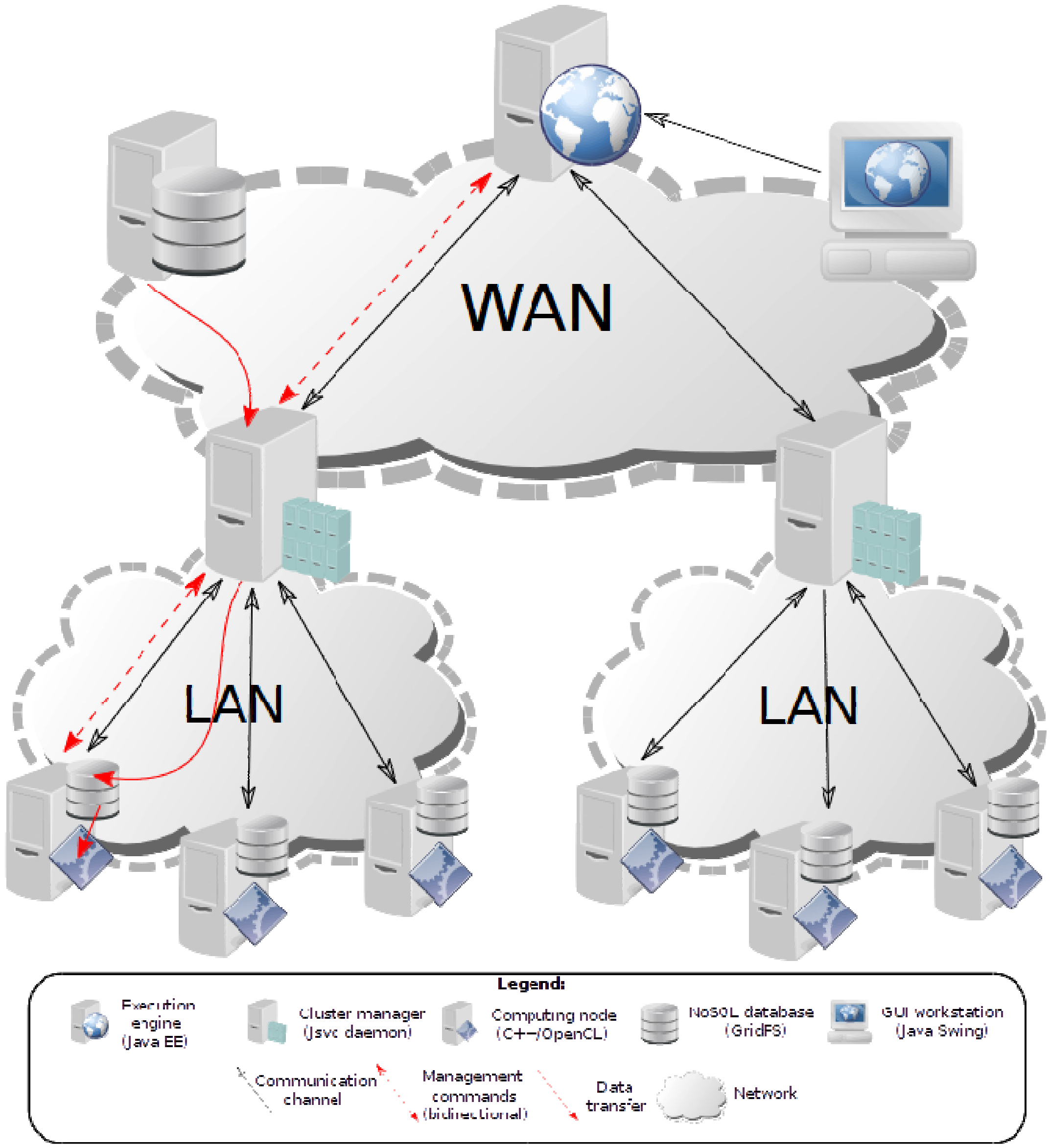

Figure 5. Modified Architecture of the Parallel System

4.1. Experiments on a Single Device

We started with testing the solution on a basic setup with one machine equipped with one Intel Core i5 processor. The execution times are shown in Figure 6. As it turns out, the results in case of a single device are as expected: for one data package, the difference between execution time with and without prefetching is negligible. The scenario of execution is the same in both cases. The more data packages, the higher the speedup of the prefetching version, reaching 30% in case of 4 input packages. The difference is significant and increasing, because in the prefetching scenarios, data transmission and computations are overlapping.

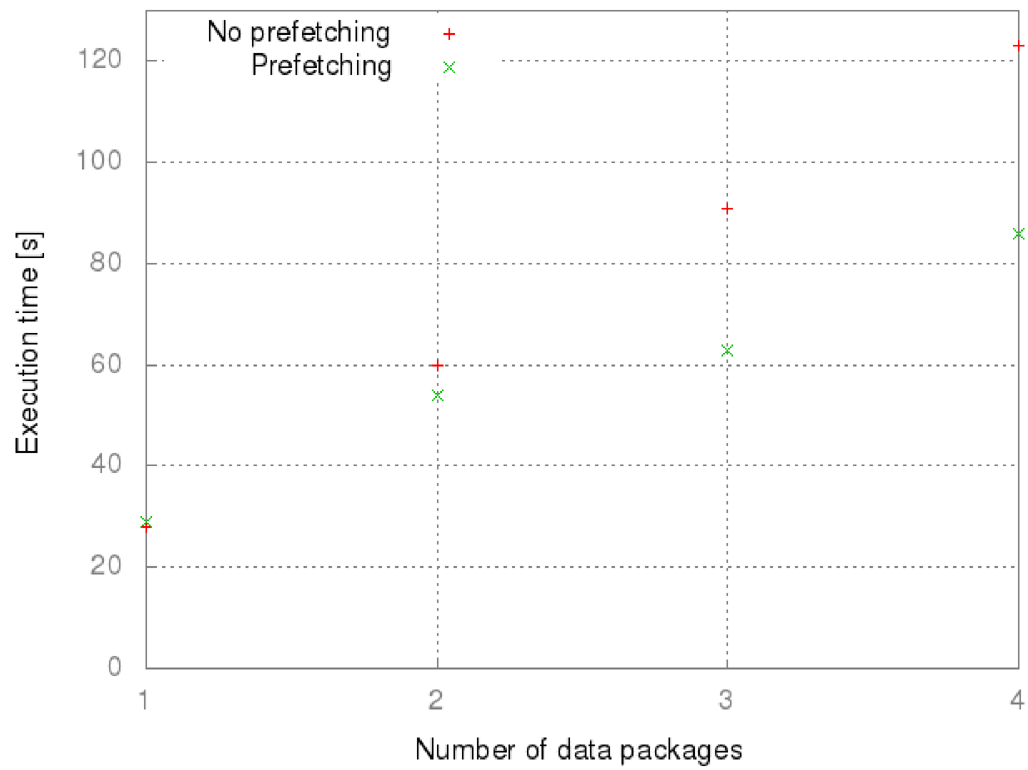

Figure 6. Prefetching Experiment Results on a Single Device Environment

4.2. Experiments on a Heterogeneous Infrastructure

After testing the proposed design in action and proving the usefulness of the prefetching optimizer, we tested the same application on a cluster of nodes equipped with different types of devices. The infrastructure for this extended tests is shown on Figure 7, which is actually a screenshot from the hive-gui application, that enables generating the infrastructure charts based on the data from the *Engine*.

In order to compare the results in the new testbed configuration to the previous ones, we had to run the application with package numbers N times higher, where N is the number of computing devices.

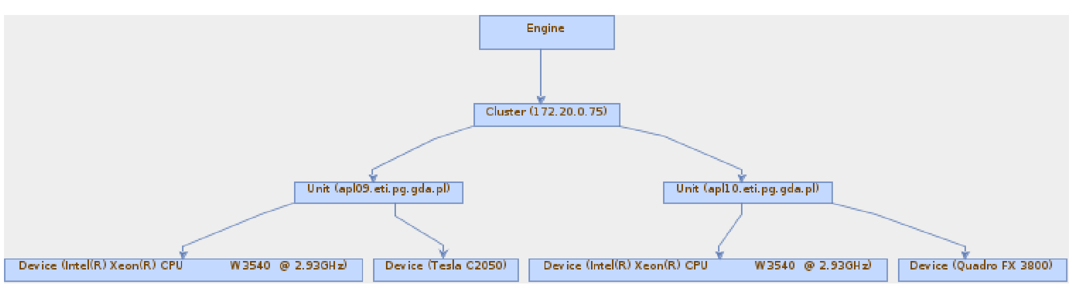

Figure 7. The Heterogeneous Testbed Configuration

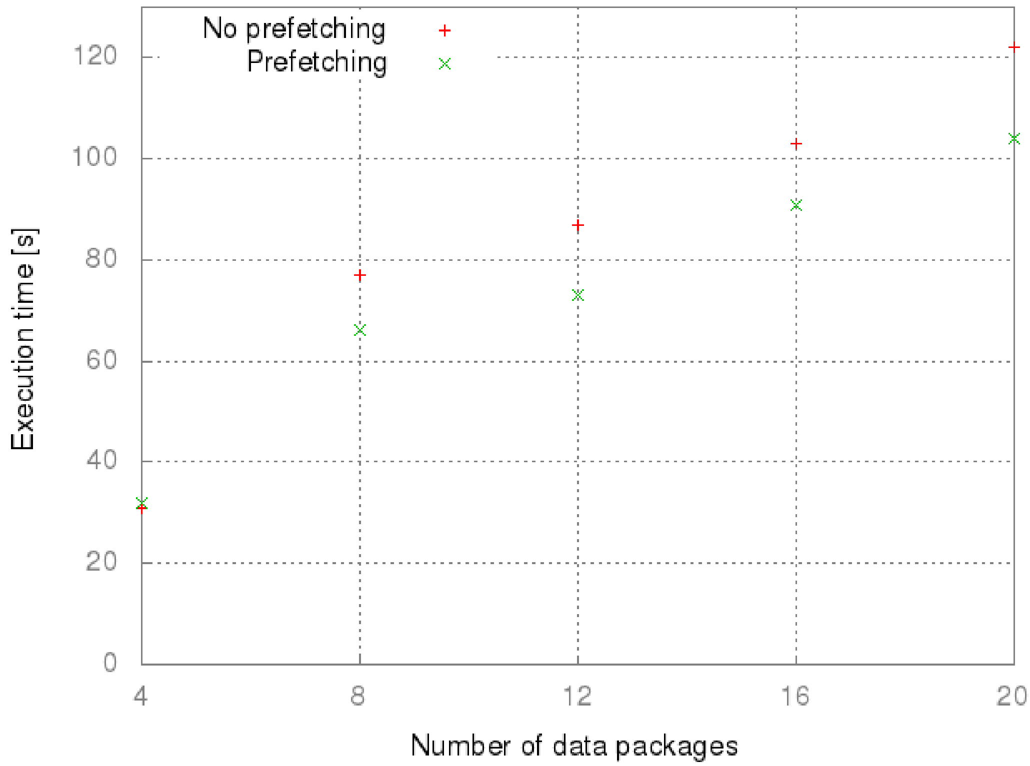

Figure 8. Prefetching Experiment Results on a Heterogeneous Environment

The results of the experiment (Figure 8) show, that the benefit from prefetching, though significant and increasing, is lower than for one device and in case of 4n packages reaches 11%. Such results could be an effect of sharing the network between multiple prefetching tasks. Still, the optimizer shows promising results in a heterogeneous environment.

4.3. Execution Simulations for Evaluating the Network-Aware Scheduler

A prototype version of the large-scale HPC system simulator described in [18] was used for the simulations in this section. The computations were modeled as a master-slave application with master denoting the KernelHive engine which distributes the data and slaves corresponding to the computing machines. To present our idea clearly, we made the following assumptions:

- tbe input data packages are of the same size;
- the overlapping of computations occurs only in the local (cluster) networks;
- the clusters are equipped with the same numbers of machines with identical performance.

Figures 9 and 10 present comparisons of execution times between various hardware configurations. The units of the y-axis are not given, because for the simulations we used a random value which is irrelevant. Proportions of the execution times are the point of the charts.

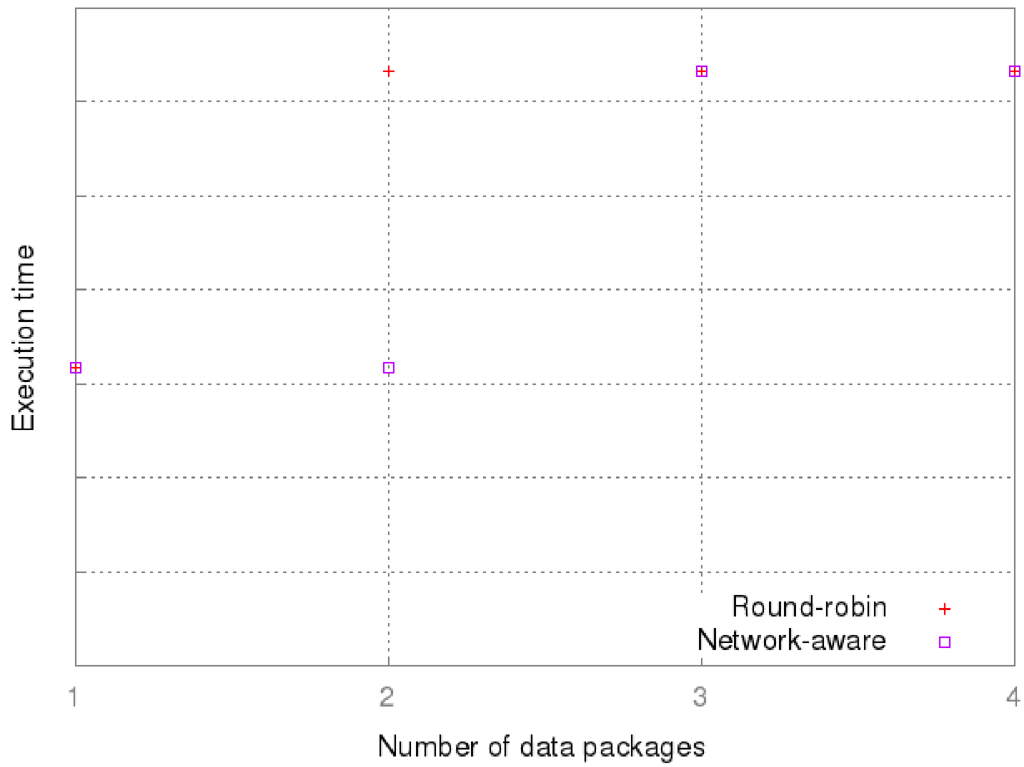

Figure 9. Simulated Scheduler Comparison for 2 Clusters with 2 Machines

Figure 9 shows the results of the simulations for a small hardware configuration consisting of two clusters with two machines each. As we can see, the difference in execution times between the round-robin and network-aware schedulers is present only in case of 2 data packages. This is the only case where it is possible to schedule each task to a machine in a different cluster. For one, three and four packages it does not make a difference where they would be placed. However, in this one case of two input data packages, the execution time of the network-aware solution is two times lower.

The aim of the simulator used for the experiment is to allow comparing various attributes of a HPC system with possibility to construct complex large-scale infrastructures. In our case it was worth estimating, how the network-aware scheduler would affect the execution times in case of bigger systems. In Figure 10 we present the results of simulations for various “M x C” configurations, where M stands for number of machines in each cluster and C stands for number of clusters.

It can be seen, that in case of larger computing infrastructures using the proposed network-aware scheduling scheme can be significantly beneficial. It should be noted, that these results are relevant only in case of HPC applications, that do not require communication between the tasks and if the assumptions listed above are met.

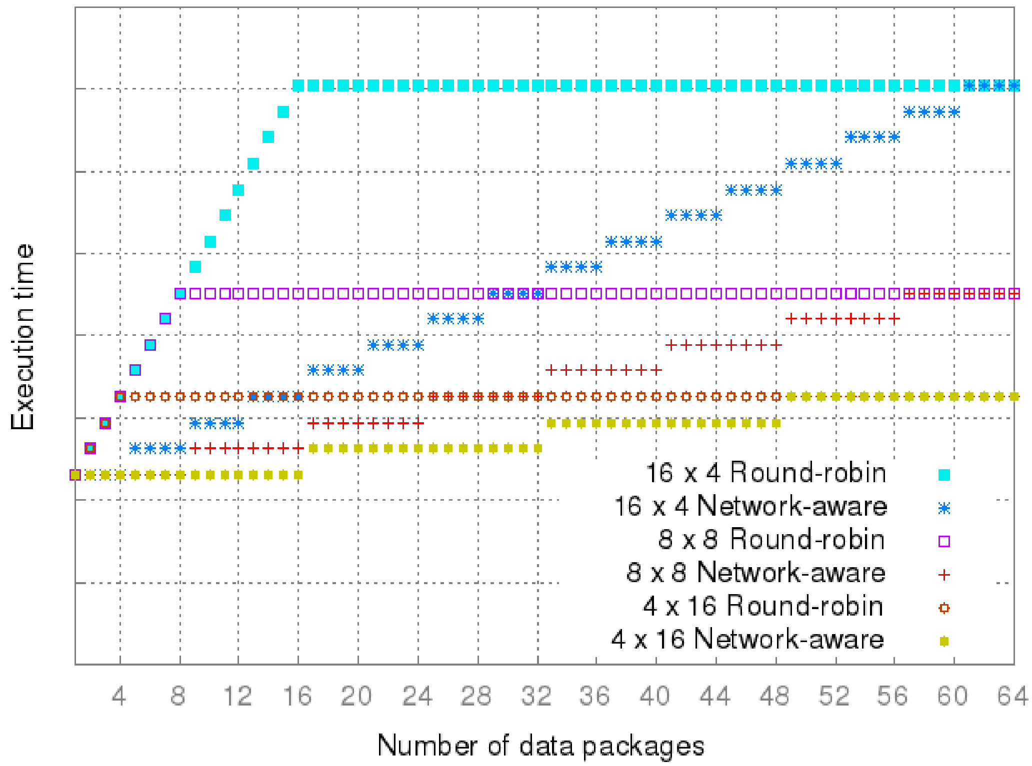

Figure 10. Simulated Scheduler Comparison for Various Environments

5. Summary and Future Work

Focusing on the aspect of data management in parallel computing systems brings up a number of issues, especially in case of heterogeneous multi–level systems. In this paper we addressed a subset of those issues by extending our parallel framework KernelHive.

We proposed an architecture with multiple distributed data servers and a versatile data addressing scheme that enables using various data storage technologies and high–level optimizations. On this basis we used GridFS as a data storage engine and presented the implementation of a new optimizer for KernelHive, that enables prefetching data to the computing devices, causing the overlapping of computations and communication and hence, reduction of execution time Our experiments, based on a regular expression matching application showed that the proposed solution is a good base for new data management schemes. Additionally, we extended our solution with a network-aware scheduling scheme and presented possible benefits from employing it into the system by performing a set of simulations using a large-scale HPC system simulator.

In the future we could extend this solution by mechanisms of dynamic transferring of intermediate results between the parallel processes with regard to their distribution, possibly among distant clusters. The proposed network-aware scheduling scheme should be verified in real parallel execution on large-scale computing systems.

ACKNOWLEDGEMENTS

Project partially supported by the grant "Modeling efficiency, reliability and power consumption of multilevel parallel HPC systems using CPUs and GPUs" sponsored by and covered by funds from the National Science Center in Poland based on decision no DEC-2012/07/B/ST6/01516. Special thanks to Tomasz Boiński (http://tboinski.eti.pg.gda.pl) for his invaluable support.

REFERENCES

[1] Grandinetti, L., ed.: Grid computing. Elsevier, Amsterdam [u.a.] (2005)

[2] Thompson, C.J., Hahn, S., Oskin, M.: Using modern graphics architectures for general-purpose computing: a framework and analysis. In: MICRO, ACM/IEEE (2002) 306–317

[3] Anderson, D.P.: Volunteer computing: the ultimate cloud. ACM Crossroads 16(3) (2010) 7–10

[4] Czarnul, P., Lewandowski, R., Rościszewski, P., Schally-Kacprzak, M.: Multi-level parallelization of computations using clusters with gpus (2013) Poster presented at GPU Technology Conference, San Jose, USA.

[5] Czarnul, P., Rościszewski, P.: Optimization of execution time under power consumption constraints in a heterogeneous parallel system with gpus and cpus. In Chatterjee, M., Cao, J.N., Kothapalli, K., Rajsbaum, S., eds.: ICDCN. Volume 8314 of Lecture Notes in Computer Science., Springer (2014) 66–80

[6] Chodorow, K., Dirolf, M.: MongoDB - The Definitive Guide: Powerful and Scalable Data Storage. O'Reilly (2010)

[7] Thompson, K.: Regular expression search algorithm. Communications of the ACM 11(6) (June 1968) 419–422

[8] Pao, D., Or, N.L., Cheung, R.C.C.: A memory-based nfa regular expression match engine for signature-based intrusion detection. Computer Communications 36(10-11) (2013) 1255–1267

[9] Kacala, B., Sagadyn, G., Sidorczak, P., Czarnul, P.: Parallel simulation of the galaxy with dark matter using gpus and cpus (2013) Poster presented at GPU Technology Conference, San Jose, USA.

[10] Khronos OpenCL Working Group: The OpenCL Specification, version 1.0.29. (8 December 2008)

[11] Bhardwaj, D., Sinha, M.: Gridfs: highly scalable i/o solution for clusters and computational grids. IJCSE 2(5/6) (2006) 287–291

[12] Edlich, S.: NOSQL Databases (January 2011)

[13] KernelHive website, viewed on July 2014, http://kask.eti.pg.gda.pl/en/projekty/kernelhive/.

[14] Rościszewski P.: Dynamic data management among multiple databases for optimization of parallel computations in heterogeneous hpc systems, in: N. Meghanathan, D. Nagamalai (Eds.), ICCSEA2014, Vol. 4 of Computer Science & Information Technology, AIRCC, 2014, pp. 9–19.

[15] Bhat, P. B., Prasanna, V. K. & Raghavendra, C. S. (1999). Adaptive Communication Algorithms for Distributed Heterogeneous Systems.. *J. Parallel Distrib. Comput.*, 59, 252-279.

[16] Ammar, R., Rajasekaran, S., Hussein, A. & Hamdy, A. (2007). Efficient Scheduling of Real-time Tandem Task Graphs on Heterogeneous Clusters with Network Limitations.. *ISCC* (p./pp. 227-232), : IEEE.

[17] P. Czarnul, Ed., Modeling Large-Scale Computing Systems. Concepts and Models. Gdansk University of Technology, 2013, iSBN 978-83- 938367-0-3.

[18] Rościszewski, P., Cychnerski, J., Brzeski, A.: A Regular Expression Matching Application with Configurable Data Intensity for Testing Heterogeneous HPC Systems, 35th International Conference Information Systems Architecture and Technology, Szklarska Poręba, Poland, 2014.