doi: 10.1093/comjnl/bxaa187

Optimization of Data Assignment for Parallel Processing in a Hybrid Heterogeneous Environment Using Integer Linear Programming

Tomasz Boiński and Paweł Czarnul*

Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology, Poland
Corresponding author: pczarnul@eti.pg.edu.pl

In the paper we investigate a practical approach to application of integer linear programming for optimization of data assignment to compute units in a multi-level heterogeneous environment with various compute devices, including CPUs, GPUs and Intel Xeon Phis. The model considers an application that processes a large number of data chunks in parallel on various compute units and takes into account computations, communication including bandwidths and latencies, partitioning, merging, initialization, overhead for computational kernel launch and cleanup. We show that theoretical results from our model are close to real results as differences do not exceed 5% for larger data sizes, with up to 16.7% for smaller data sizes. For an exemplary workload based on solving systems of equations of various sizes with various compute-to-communication ratios we demonstrate that using an integer linear programming solver (lp_solve) with timeouts allows to obtain significantly better total (solver+application) run times than runs without timeouts, also significantly better than arbitrary chosen ones. We show that OpenCL 1.2's device fission allows to obtain better performance in heterogeneous CPU+GPU environments compared to the GPU-only and the default CPU+GPU configuration, where a whole device is assigned for computations leaving no resources for GPU management.

Keywords: parallel computing; data assignment; hybrid heterogeneous environment; integer linear programming.
Received 30 July 2020; Revised 2 November 2020; Accepted 24 November 2020
Handling editor: Antonio Fernandez Anta

1. INTRODUCTION

In today's high-performance computing systems, heterogeneity and hybridity have become present and widespread at various levels, including both hardware and software levels. Practically, most of the modern workstations, servers as well as cluster nodes feature multi- or many-core CPUs as well as accelerators such as GPUs. Making the most of the available computational power requires knowledge of and ability to combine various APIs such as OpenMP, OpenCL for parallelization among CPU cores within a node, CUDA, OpenCL or OpenACC for GPUs, MPI for inter-node parallelization [8]. This brings along classical challenges [12] such as scheduling that is associated with initial data partitioning as well as subsequent selection of compute devices and data mapping for computations taking into account a given optimization goal such as minimization of execution time [8], minimization of execution time with a bound on power consumption [14], energy-aware goals [13], etc. In this paper, we consider the aspect of data partitioning, scheduling and its optimization in such an environment using a mixed -integer linear programming method and analyzing its practical complexity for a heterogeneous environments with compute devices of various generations (that naturally arise out of system upgrades) and drawing practical conclusions for programmers, especially in the context of efficient usage of OpenCL in a heterogeneous CPU+GPU environment.

Evolution of heterogeneous computing systems, the approaches and challenges in high-performance computing emphasize constant need to investigate data partitioning and scheduling in such systems [12, 34]. This is further justified

by emergence of many applications running on CPU+GPU systems, from various domains, such as parallelization of large vector similarity computations [9], data stream processing and its optimizations including numbers of computational CPU threads, number of CUDA streams, overlapping computations and communication, energy aspects of CPU+GPU processing [10], a microscopy image analysis application [37] including tests using CPU cores, MIC and GPU, a hybrid pattern matching algorithm for deep packet inspection [22], etc.

2. RELATED WORK AND MOTIVATIONS

2.1. Selected frameworks allowing parallelization in heterogeneous high-performance computing systems

As today's high-performance computing systems offer the possibility of parallelization at various levels, proper APIs are available for implementation of communication and synchronization. At the inter-node level within a cluster or generally between nodes the main representative is Message Passing Interface (MPI); for multi- and many-core CPUs: OpenMP, OpenCL and Pthreads; and for GPUs: CUDA, OpenCL and OpenACC. Combinations of those approaches for hybrid applications are also widely used [12]. As from the implementation point of view, this paper relies on a framework implemented with MPI and OpenCL, works including similar solutions are of interest, technology-wise.

The book by Czarnul [8] includes several generic implementations for key programming paradigms, such as master-slave, geometric Single Program Multiple Data (SPMD) as well as divide-and-conquer. Ready-to-use templates are provided, also for hybrid implementations including: MPI+OpenMP, OpenMP+CUDA and MPI+ CUDA codes. Optimizations including data prefetching, minimization of synchronization overheads, load balancing and impact of data granularity on performance are discussed. In [25], a survey of techniques, especially for CPU+GPU heterogeneous computing systems, was presented. Workload partitioning for such systems is discussed along with APIs and languages supporting development of codes. Support at levels such as compiler and framework is described. Techniques for energy saving (such as workload partitioning and DVFS) as well as benchmark codes for evaluation of such systems are given as well. As this field is developing very rapidly, new approaches and techniques are being continuously developed. In this respect, [12] can be seen as a more up-to-date review of parallelization approaches while [13] as a more-up-to-date review of energy aspects for high-performance computing systems.

There are several frameworks and systems which allow parallelization of computations in hybrid high-performance computing environments.

SkePU is a C++ framework meant for heterogeneous systems focusing on multi-core and systems with potentially many accelerators. Öhberg et al. [27] proposes and discusses performance of a back-end for SkePU version 2 [15] able to schedule a workload on CPU+GPU systems and showing better performance than the previous SkePU implementation. SkePU uses several skeletons such as Map, Reduce, MapReduce, MapOverlap, Scan and Call. Original implementation of SkePU 1 used StarPU [1] as back-end for heterogeneous systems. StarPU is task based that encompasses input data along with codelets that can be written with C/C++, CUDA, and OpenCL. Scheduling policies such as eager, priority and random along with caching are available. The new back-end in SkePU 2 works with both CUDA and OpenCL. The workload is partitioned among CPU and accelerator spaces based on a ratio set manually or determined automatically. Partitioning of the aforementioned skeletons is presented in [27]. A simple linear performance model is built, that distinguishes CPU and accelerator components. This was used for autotuning by the system. The authors demonstrated speed-ups of the hybrid auto-tuned versions over CPU and accelerator versions.

KernelHive is a system that uses a multi-level architecture that allows spanning computations across several clusters or LANs, across several nodes within a cluster or a LAN and across several compute devices of various types within each node. Upper management layers: hive-engine selecting compute devices based on optimization criteria as well as managing computations and hive-cluster managing computations on each cluster, are implemented in Java. hive-unit and hive-workers located at node level were implemented in C++. Upper-most layer hive-gui allows a user to define an application as a workflow graph with vertices corresponding to operations such as data partitioning or merging or computational functions defined in OpenCL for portability across various compute devices including CPUs and GPUs. Data is downloaded from indicated data servers. This architecture allows for easy substitution of optimizers that can be performance, performance-energy oriented, or custom built by the user. Demonstration of system's scalability in a heterogeneous CPU+GPU environment (up to 18 nodes) was presented in [31] for a parallel MD5 password-breaking application. The overhead of up to 11% of KernelHive for inverse distance weighing interpolation of geospatial data was measured compared to pure MPI+OpenCL code.

In [40], authors described an OpenCL task scheduling method which schedules multiple kernels from multiple programs in a CPU+GPU heterogeneous platform. It predicts utilization of a device by a kernel. The authors showed that speedup can be a good scheduling priority function and presented a model predicting a kernel's speedup based on its code structure. Such prediction and input data size are used for scheduling of a large set of concurrent OpenCL kernels. The authors improved, compared to other approaches, throughput 1.21 and 1.25 times on tested NVIDIA and AMD systems while turnaround time was improved 1.5 and 1.2 times, respectively.

TREES [20] is a task-parallel runtime system, meant for execution on CPU+ GPU systems, implemented with OpenCL. It utilizes a work-together approach that follows the following principles: the overhead on the critical path should be spent by the whole system at once and work overheads should be spent cooperatively. The paper shows, in particular, reaching 0.5-0.7 performance of native OpenCL for sorting and promising performance of TREES on GPUs compared to cilk running in parallel on CPUs, assuming low OpenCL overheads for future CPU/GPU chips.

Quasar [18] is a programming framework, aimed at image and video processing, that can run in a hybrid CPU+GPU environment. It provides a programmer with a programming language and an IDE, a memory manager, scheduler, load balancer (managing compute devices) as well as a runtime back-end supporting CUDA and OpenCL. Furthermore, visualization via OpenGL is possible. The environment allows easy prototyping and execution. In [19], comparison of execution times is given for 7 benchmarks implemented using OpenACC, CUDA and Quasar with comparable times of the latter two with Quasar codes being approximately 0.5 and 0.25 of the former two in terms of length.

In [36], it was shown that highly specialized codes such as 3D stencil computations can be efficiently transformed, with use of proper additional directives, into efficient parallel codes utilizing a combination of APIs such as MPI+CUDA+OpenMP using the proposed source-to-source compiler. The authors have presented the capability to achieve around 90% of performance of highly, manually tuned codes demonstrating a good trade-off between performance and ease of development for the solution.

PSkel [28] is another framework allowing parallelization within a CPU+GPU environment targeted at stencil type computations. The solution is based on parallel skeletons and using Intel TBB and NVIDIA CUDA as back-ends. Authors demonstrated speed-ups of the CPU+GPU version up to 76% and 28% as compared to CPU-only and GPU-only versions respectively.

Other specialized applications include query processing and necessary adaptation of patterns and techniques in order to achieve better times compared to CPUs for 6 out of 7 TPC-H queries [29].

In [33], authors show a framework that is able to use multiple CPUs and GPUs simultaneously for parallel and efficient computing of k-vertex induced sub-graph statistics, i.e. graphlets. Specifically, dynamic load balancing and complexity are discussed. It has been demonstrated that the proposed hybrid CPU+GPU approach using Intel Xeons + NVIDIA GTX Titans is considerably better than both CPU and GPU only results.

Demonstration of a computational framework for parallelization across CPUs and GPUs is presented in [4], for flow simulation type of processing. Specifically, a two blade hovering rotor is investigated in terms of performance. It is interesting to note that the authors obtained best times with a GPU-only configuration with 24 GPUs, a CPU-only configuration with 126 CPU cores. They note the need for the number of cores for GPU management, that is considered in the model and assessed in terms of performance compared to CPU only and GPU only results in this very work.

It should be noted that apart from selection of a proper combination of APIs considering parallelization levels that are to be exploited by a parallel application [8], typically several low-level optimizations can be applied in order to minimize execution time of a hybrid CPU+GPU application. These include, in particular: setting a proper number of computational threads on CPU cores such that there are cores left for host threads managing computations on GPU(s), overlapping communication and computations using at least 2+ CUDA streams (or queues in OpenCL) [10], reducing GPU offload latency via fine-grained CPU-GPU synchronization with F/E (Full/Empty) bits [24], optimizing CPU-GPU command offloading for real-time applications using Vulkan [3].

Furthermore, there exist approaches, proposed for use at various abstraction levels, that allow simulation of processing on CPU+GPU systems. For instance, in [39] authors proposed Multi2Sim, a simulation framework that provides architectural simulation for x86 CPUs and Evergreen GPUs, performed at the ISA level. Errors between the simulation and real execution times for benchmarks such as BinomialOptions, BitonicSort, DCT, MatrixMultiplication, SobelFilter, URNG, ScanLargeArrays, RadixSort were reported between 7 and 30% with an average of approximately 20%.

MERPSYS offers modeling of cluster nodes with CPU and GPU components with selection of devices from its database along with definition of cluster models with nodes and network interconnects. This is done through a visual editor that allows to create models of systems composed of a very large number of components. Applications are modeled in Java extended with MPI-like methods modeling communication and synchronization. This abstraction level allows simulation of applications reflecting shared memory programming on CPUs, CPU+GPU systems as well as parallelization among cluster nodes. The system allows simulation of application execution time along with energy consumed during execution. In [11], simulation and modeling results using CPUs for a master-slave application for up to 64 processes with an average error of 1.4% and a maximum error of 4.7%, geometric SPMD application up to 1000 processes with an average error of 5.4% and a maximum error of 23.3% and a divide-and-conquer application up to 1024 processes with an average error of 4.8% and a maximum error of 17.8% was presented. In [17], modeling running a saxpy GPU kernel with configurable compute intensity with simulation results showing an average error for large data size 3.7% (GTX 970), 4.6% (GTX TITAN X) and 13.6% (GTX 1070), compared to real runs was presented. In [32], results of modeling and simulation of deep neural network training in a CPU+multiple GPU environment with a mean

percentage error up to 2.7%, regarding execution times, was presented.

2.2. Integer (non-)linear programming approaches in the context of high-performance heterogeneous systems

In [6], authors focused on optimization of task scheduling which minimizes energy usage with a specified computation's deadline for the class of stencil computations. Single node configurations with heterogeneous processors were assumed which corresponds to modern workstations with CPUs and GPUs. An integer linear programming approach was taken and investigated. Accuracy of the model was shown to exceed 90%. Four heuristics have been proposed and compared to optimal ILP results. The authors concluded that for CPU+GPU environments, proper mapping of stencil tasks to the underlying system considering communication costs plays an important role in minimization of execution time and energy cost.

In [35], authors proposed two algorithms using Integer Programming (IP) solvers for job scheduler SLURM with code implemented as plug-ins. Furthermore, they verified the possibility to use such a setting in a SLURM emulation mode, ready to use in production for a large number of variables, such as even 150K. According to the paper, the proposed AUCSCHED algorithm allows to obtain better utilization, spread and packing, the proposed IPSCHED obtains better waiting time, as compared to SLURM Backfill which gives better results in terms of fragmentation.

Authors of paper [2] focused on algorithmic optimizations of scheduling independent and moldable tasks on multiple CPUs and multiple GPUs. Specifically, they presented a dual approximation scheme which incorporated a fast integer linear program (ILP). The task was two-fold, i.e. determination of mapping a task to a CPU(s) or a GPU and finding out the number of CPUs to assign, in the former case. An approximation ratio of $3/2 + \varepsilon$ is shown along with presentation of a polynomial-time algorithm with an approximation ratio of $2 + \varepsilon$. The presented approach was shown to be faster than a modified HEFT algorithm. One of the goals of that research was to show that an ILP-based algorithm is able to solve larger problem instances in a reasonable amount of time.

Mixed integer linear programming has also been used in other, similar contexts such as for scheduling workflow applications [7]. In this case, a workflow is represented by an acyclic directed graph in which vertices correspond to tasks that need to be executed and edges denote dependencies. For each task, there is a set of services, with various parameters such as cost and execution time, out of which one has to be selected for each task. Optimization criteria can include minimization of whole workflow execution time with a bound on the cost of selected services, minimization of execution-time product, etc. In [7] it was shown, through several experiments, that for workflows of type epigenomics, or sequences of the epigenomics workflows, workflow applications for processing of multimedia content such as application of successive filters on digital images, business workflows for a company subcontracting services or buying components from others, the MILP approach proved better than divide-and-conquer, GAIN and genetic approach, for minimization of workflow execution time with a bound on total service cost. Particular services may have execution time/cost values corresponding to parameters of various compute devices in a heterogeneous environment while edges allow for various communication costs for various combinations of services, in the ILP formulation.

In [5], authors formulated an MILP model for an unrelated parallel machines scheduling problem with consideration of penalty cost of makespan and time varying electricity cost. Benefits of the approach compared to genetic algorithm based approaches were shown.

The authors in [26] discuss the impact of various mixed-integer linear programming formulations for parallel machine scheduling problems with consideration of, in particular, total weighted completion time, makespan, maximum lateness, total weighted tardiness. Formulated conclusions allow to prefer particular formulations in terms of relative completion times, frequency of obtaining optimal solution, problems with non-zero job release dates as well as sizes of job processing times.

In [23], authors proposed to use Mixed Non-linear Integer Programming (MNILP) to distribute input data among a collection of herogeneous GPUs, based on prior profiling. They presented good results compared to uniform distributed and distribution based on performance estimates using numbers of cores multiplied by clock frequency. However, experiments only included small environments with 4 GPUs. Compared to that work, our approach considers a much more complex system, also with Intel Xeon Phis and communication costs.

2.3. Motivations

In view of the aforementioned works, CPU+GPU environments are important, in many labs there are heterogeneous CPU+GPU environments with compute devices of various generations. Consequently, it is important to develop an approach to target scheduling applications for such environments and provide generic guidelines for optimization. In this paper we provide these for a model of an application processing a set of data chunks in parallel, with configurable compute-to-communication time ratio.

The main contribution of the proposed model and this paper is as follows:

- We assess accuracy of the model in a real-world heterogeneous parallel system as well as obtain model coefficients corresponding to a real CPU+GPU system taking into account results of the aforementioned analysis. Specifically, we take into account various overheads that show up in a multi-level parallel system, i.e. startup times and bandwidths not only between nodes in a parallel

system but also corresponding to communication within a node using, e.g. PCI Express, initialization of OpenCL stack, etc.

- We analyze computation performance in terms of hardware discrepancy. This allows us to observe overheads corresponding to various system components including, e.g. impact of co-scheduling computations on a CPU and a GPU within the same node or hardware disparity, i.e. various performance coefficients for identical hardware (like two GPUs) placed in various PCI Express slots. The hardware load impact is also considered for proper scheduling of computations using OpenCL's subdevices (so called device fission) versus direct scheduling the computations straight on the default devices present within the node.
- We investigate the actual complexity of the scheduling problem for a typical LAN based heterogeneous environment with workstations as well as servers with compute devices of various generations. This is performed by using various timeouts for the scheduling phase in the context of the quality of the schedule versus its running time. This, in turn, allows us to optimize application running time, also considering GPU-only, as well as CPU+GPU configurations. Results of tests are shown for a heterogeneous environment and various problem sizes corresponding to various compute-to-communication ratios.

The proposed model takes into account all aspects of code execution including data preparation times, function call times and data transfer times, both between nodes and within the node to its Processing Units (e.g. GPU). Furthermore, the model can be easily extended with multiple layers – in our test we consider a manager-worker configuration where the data is transferred between the manager and worker nodes and than within worker nodes to processing units. The worker node can be, e.g. a cluster and then, by recursion, the model can consider proper data assignment taking into account the transfer between the manager node and worker cluster and then to cluster nodes playing the role of cluster's Processing Units. The distribution with the cluster itself can be calculated analogously.

3. PROCESSING MODEL AND APPROACH TO SOLUTION

For the considered model, let us adopt the following notation for input, known data:

N_i – node N_i is the i-th node in the environment,
$P_{PU_j^{N_i}}$ – performance of the j-th PU (Processing Unit) on node N_i counted as the number of problems solved per time unit, where as a problem we consider processing of one data packet,
d_{in} – size of input data,
d_{out} – size of result,
d_{si} – size of a single input data packet,
d_{sr} – size of a single result data packet.

Let us adopt the following notation for variables for which values need to be found:

$d_{PU_j^{N_i}} \in N_+$ – numbers of packets assigned to the j-th PU (Processing Unit) on node N_i.

The following condition must be satisfied for the input data:

$$\sum_i \left(d_{si} \cdot \left(\sum_j d_{PU_j^{N_i}} \right) \right) = d_{in} \tag{1}$$

and for the result:

$$\sum_i \left(d_{sr} \cdot \left(\sum_j d_{PU_j^{N_i}} \right) \right) = d_{out} \tag{2}$$

Using the adopted notation execution time of the whole application can be expressed as:

$$\begin{aligned} T = t^{gp} + \max_{N_i}((a_{N_i} + \frac{d_{si} \cdot d_{N_i}}{b_{N_i}}) + \\ t^p_{N_i} + \max_j (d_{PU_j^{N_i}} \cdot ((a_{PU_j^{N_i}} + \frac{d_{si}}{b_{PU_j^{N_i}}}) + \\ t_{PU_j^{N_i init}} + \frac{1}{P_{PU_j^{N_i}}} + t_{PU_j^{N_i dinit}} + (a_{PU_j^{N_i}} + \frac{d_{sr}}{b_{PU_j^{N_i}}}))) \\ + t^m_{N_i} + (a_{N_i} + \frac{d_{sr} \cdot d_{N_i}}{b_{N_i}})) + t^{gm} \end{aligned} \tag{3}$$

where:

- d_{N_i} is the number of packets assigned to node N_i: $d_{N_i} = \sum_j d_{PU_j^{N_i}}$;
- t^{gp} denotes time spent (if needed) on input data partitioning;
- t^{gm} denotes time spent (if needed) on result merging;
- $t^p_{N_i}$ denotes potentially (if needed) additional data partitioning time within a node;
- $t^m_{N_i}$ denotes potentially output merging time within a node such as integration of results from GPUs and CPUs;
- $t_{PU_j^{N_i init}}$ denotes time needed to prepare and start the calculation function, it includes the parameters transfer via system bus, system call time, etc.;
- $t_{PU_j^{N_i dinit}}$ denotes time needed for cleanup after and return from the calculation function, it includes the result transfer via system bus, system call return time, etc.;

a_{N_i}, b_{N_i}, $a_{PU_j^{N_i}}$ and $b_{PU_j^{N_i}}$ are coefficients that correspond to effective startup time (a_*) and bandwidth (b_*) when communicating between the manager node and node N_i or with a processing unit (i.e. over PCI Express) inside node N_i respectively. Effective means that values of these coefficients may vary depending on actual implementation, i.e. whether it uses standard CPU-GPU communication, invocation and GPU-CPU communication or overlapping communication and computations [10] through use of many streams (CUDA) or command queues (OpenCL).

We should note that we profile execution time of each packet as $\frac{1}{P_{PU_j^{N_i}}}$ which allows us to consider it in the aforementioned formula even if the processing time of a packet is not a linear function of d_{si}.

Subsequently, an optimization goal needs to be specified. Within this paper we consider minimization of execution time T. Consequently, optimization in this case requires finding values of $d_{PU_j^{N_i}}$ variables, i.e. data assignment to compute units need to be found in order to minimize application execution time.

When the expected data scheduled for one of the nodes exceeds the memory capacity of that node the model can be extended with a limit stating the maximum data size that should be sent to the node in question. Failing to do so, as discussed in Section 4.5.2, can lead to an undesired scenario, as going over the memory capacity greatly increases kernel initialization and deinitialization times.

The optimization methodology, for a given application, includes performing the following steps: obtaining a_* and b_* values through profiling, having the a_* and b_* values use an integer linear programming solver to find desired data assignment configurations, investigating optimal versus heuristic approaches, including solver running time and final application execution time.

4. EXPERIMENTS

4.1. Testbed environment

For the tests we decided to use a part of the environment available at the Department of Computer Architecture at the Faculty of Electronics, Telecommunication and Informatics, Gdansk University of Technology, Poland. The environment consists of machines obtained during different time periods, for various types of projects. It is also a semi-open environment, where multiple users can run different programs consuming selected resources. As such, it suits our purpose and can be treated as a heterogeneous environment not dedicated to a given problem. For the tests we decided to use the following machines:

- two servers (called *apl09* and *apl10*) with Intel Xeon W3540 cpu running at 2.93GHz clock speed, 4 cores with hyper threading each (OpenCL platform 1, device 0), 12GB of RAM, NVIDIA Corporation GF114 GeForce GTX 560 Ti (OpenCL platform 0, device 1 on both *apl09* and *apl10* servers, platform 0 device 0 is not used in the tests as in both cases it was occupied with very old hardware) GPU, with CentOS 7.4 Linux operating system,
- server (called *apl11*) with two Intel Xeon E5-2640 CPUs running at 2.50GHz clock speed, 6 cores with hyper threading each (OpenCL platform 1, device 0), 64GB of RAM and two NVIDIA Corporation GK110GL Tesla K20m GPUs (OpenCL platform 0, devices 0 and 1), with CentOS 7.4 Linux operating system,
- server (called *apl12*) with two Intel Xeon E5-2680 CPUs running at 2.80GHz clock speed, 10 cores with hyper threading each (OpenCL platform 0, device 0), 128GB of RAM and two Intel Corporation Xeon Phi coprocessors 5100 (OpenCL platform 0, devices 1 and 2), with CentOS 6.9 Linux operating system.

All servers are connected via a 1Gbit/s Ethernet network. Due to the technical reasons (operating system and library versions that could not be upgraded) we selected *apl12* server as the manager node that will perform both calculations and will distribute tasks to other servers.

4.2. Application framework and testbed application

4.2.1. *Application framework*

The testbed framework is written in C and uses MPICH 3.1 for inter-process communication and OpenCL 1.2 for calculations. We have chosen OpenCL as a general framework for computations as it allows running the same code in a heterogeneous environment where creation of dedicated implementations might have been time consuming. The main application is run on the *apl12* server and distributes tasks using MPI messages. In case of servers *apl09*, *apl10*, *apl11* and *apl12* the tasks are distributed directly to threads responsible for calculations on the CPU and each GPU or coprocessor. The architecture of the test application is presented in Fig. 1.

4.2.2. *Testbed application*

The proposed solution aims to be able to cope with various applications for which processing steps include partitioning, processing and result merging, effectively available to be modeled by acyclic directed graphs in which these steps correspond to graph nodes. The testbed environment should thus allow verification of the proposed approach for different ratios of computation to communication times. For that reason during the tests of the proposed model we decided to implement a parallel solution for solving $Ax = b$ systems of equations using the Jacobi method. Throughout the paper we will use term problem of size n to denote solving 1 system of n linear

TABLE 1. The expected time and average real time (from 100 runs) for problems of size 512.

Number of problems	64	128	256	512	768	1024	1536	2048
Time [s]	5.303	10.404	20.532	40.674	60.747	81.041	121.287	161.718
Real time [s]	6.186	11.03	22.085	41.759	63.449	82.04	123.238	167.277
Diff [s]	0.883	0.626	1.553	1.085	2.702	0.999	1.95	5.559
Diff [%]	16.65	6.018	7.562	2.667	4.449	1.233	1.608	3.438

TABLE 2. The expected time and average real time (from 100 runs) for problems of size 1024.

Number of problems	64	128	256	512	768	1024	1536	2048
Time [s]	11.939	23.464	46.101	92.03	138.038	183.982	275.75	367.474
Real time [s]	13.522	24.939	46.428	91.917	137.888	185.548	275.813	366.525
Diff [s]	1.583	1.475	0.327	-0.112	-0.15	1.566	0.063	-0.948
Diff [%]	13.256	6.287	0.71	-0.122	-0.108	0.851	0.023	-0.258

TABLE 3. The expected time and average real time (from 100 runs) for problems of size 2048.

Number of problems	64	128	256	512	768	1024	1536	2048
Time [s]	32.357	63.368	125.485	248.674	373.62	496.441	807.889	1304.797
Real time [s]	33.861	64.619	125.442	248.16	379.882	494.486	797.254	1303.96
Diff [s]	1.504	1.251	-0.043	-0.514	6.262	-1.954	-10.636	-0.836
Diff [%]	4.647	1.974	-0.034	-0.207	1.676	-0.394	-1.316	-0.064

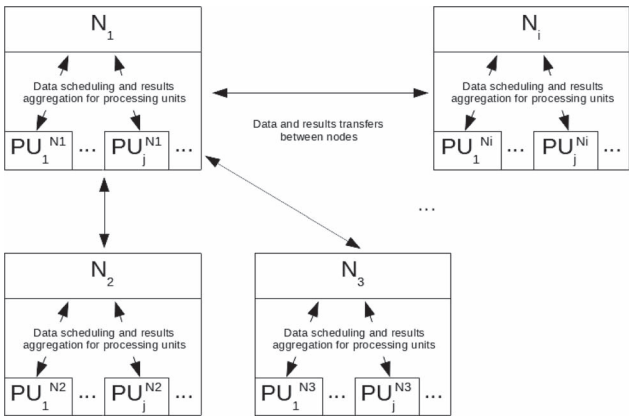

FIGURE 1. Test application architecture.

TABLE 4. Comparison of calculation times between optimal, arbitrary and proportional to Processing Unit performance configurations for 2048 problems of size 1024.

Configuration	Real time [s]
Optimal	366.525
Arbitrary 1	2680.943
Arbitrary 2	1764.693
Arbitrary 3	2598.601
Proportional	537.559
Min-min v1	726.168
Min-min v2	395.024

equations with n unknowns using the Jacobi method. The area of linear equations is vital for solving many of the current problems. In the paper, the Jacobi method is used as a workload that can be easily parametrized. By changing parameters like the matrix size (thus the number of equations) or the number of iterations we can change proportions between computation and communication times. Consequently, obtained results, in terms of scheduling, can also be useful for other workloads with similar computation to communication ratios. From the application point of view, systems of linear equations play an important role in various analysis and optimization problems. They are used in polynomial interpolation, to model electricity in circuits. Other uses can be observed in astronomy, traffic management, electromagnetics [21] and particle simulations [16]. All those fields usually deal with multiple objects that have to be tracked or modeled introducing a need for the ability to solve multiple systems of linear equations in parallel. Specifically, this can be useful when solving many problems with various input data sets in parallel which backs up the approach adopted in this paper. Additionally, the test framework, with very basic modifications, can be used for any problem following the same

TABLE 5. Comparison of calculation times between optimal, arbitrary and proportional to Processing Unit performance configurations for 2048 problems of size 2048.

Configuration	Real time [s]
Optimal	1303.96
Arbitrary 1	8946.678
Arbitrary 2	6963.740
Arbitrary 3	11996.284
Proportional	1713.970
Min-min v1	1899.181
Min-min v2	1404.234

application model. The user only needs to modify the data structures used to store data and provide a proper OpenCL kernel that will perform actual calculations.

We aimed at testing the solution with different ratios between communication and computation times. To achieve that we decided to test the approach with different matrix sizes. The tests were thus performed with matrix sizes of 512x512, 1024x1024 and 2048x2048 each, holding a double value in each cell. Single problem data sizes, using double precision, for matrix size of 512, 1024 and 2048 are 2101248, 8396800 and 33570816 bytes respectively. For 1000 problems to solve, of size 2048, the total input data size is 33570816000 bytes (around 32GB). Results are far smaller, each requires 4096, 8192 and 16384 bytes respectively. For 1000 problems of size 2048, this accumulates to 16384000 bytes (around 16MB). Due to the size of the input data, the amount of the RAM memory available on the servers and the limitations to `malloc`/`calloc` functions in C language the input data are stored in files mapped to the shared memory. Data for calculations received by each compute unit is stored in the system memory for faster access.

We decided to set the global work size equal to the size of the input matrix. The local work group size was set to 128, as this value should be divisible by 32 for CUDA devices and will allow usage of a greater number of threads on the devices that support it.

4.3. System profiling

The environment described above was profiled to determine values of P_*, a_* and b_* parameters and to identify all potential bottlenecks. Profiling was needed to assess both the calculation time on a given Processing Unit and transfer time between nodes and to the Processing Units.

Average time needed for the j-th PU on node N_i to calculate 1 iteration of Jacobi method was derived as the average time from 100 calculations for a given problem size, 1000 Jacobi iterations. Due to the fact, that the CPU was used for the calculations and for system management at the same time, an OpenCl 1.2 feature called *device fission* [38] was used by creating OpenCL sub-devices. On *apl09*, *apl10* and *apl11* servers 2 threads were dedicated to management. On *apl12*, as it served as the manager node for all calculations we decided to use 8 threads for management purposes and 32 for calculations. The detailed results are presented in Table A.1 in Appendix A. The devices are described as <node name>(<platform id>, <device id>). The tests also revealed that from the practical point of view proper profiling can be done with fewer calculations as the difference between the average and the real time does not exceed 5%. In our case for the rest of the test we used only the results obtained while calculating 200 systems of equations on any given Processing Unit.

The network and data bus profiling allowed us to measure a_{N_i}, b_{N_i}, $a_{PU_j^{N_i}}$ and $b_{PU_j^{N_i}}$ coefficients. In all cases the transfer times were calculated as an average from 1000 bidirectional data transfers. The network transfer times were done starting from *apl12* node as it was the manager node for the test framework. The a_{N_i} and $a_{PU_j^{N_i}}$ coefficients are average values for sending empty message between nodes and the node and its Processing Units. Value of b was calculated as in Equation 1. In the aforementioned formula S_{m1} and S_{m2} are sizes of two messages (in bytes) and T_{m1} and T_{m2} are times (in seconds) needed to transfer the message in one direction between given nodes or devices. For calculations we used times needed to transfer messages of sizes 0.5MiB, 1MiB, 2MiB, 8MiB, 16MiB and 32MiB. For smaller packages the transfer times were inconsistent due to the small package size. The final coefficients b_{N_i} and $b_{PU_j^{N_i}}$ were calculated as averages for the aforementioned message sizes.

$$b = \frac{S_{m1} - S_{m2}}{T_{m1} - T_{m2}} \tag{4}$$

The detailed profiling information is presented in appropriate tables in the Appendix section. The transfer times of a single package (in ms) are presented in Table A.2 and values of a_{N_i} and b_{N_i} in Table A.3. For transfers and coefficients from the host to Processing Units available see Tables A.4 and A.5 respectively.

4.4. Results

We used Equation 3 to calculate the optimal problem distribution for the environment described in Section 4.1. We used the Jacobi solver to calculate problems with 512, 1024 and 2048 unknowns. Each problem was approximated using 1300 iterations. We tested the proposed solution for distribution of 64, 128, 256, 512, 768, 1024, 1536 and 2048 problems by scheduling calculations on configurations that included CPUs, GPUs and Intel Xeon Phi devices. The expected time and average real time (from 100 runs) needed for calculations are presented in Tables 1, 2 and 3 for problems of size 512, 1024 and 2048 respectively, where Time represents the theoretical time (in seconds) returned by lp_solve, Real Time

TABLE 6. Initialization and deinitialization times with respect to percentage of RAM usage for problem of size 2048.

Server	Device	RAM usage [%]	Initialization time [s]	Deinitialization time [s]
apl09	(0, 1)	0-100	0.0684957	0.0351841
		100-106	0.4006795	0.1230411
		106-150	0.7705788	0.2866877
apl10	(0, 1)	0-100	0.0562228	0.0268629
		100-106	0.2796496	0.1239514
		106-150	0.4575722	0.1702592

TABLE 7. Profiling results (in seconds) for problems of size 1024 without the use of CPU as Processing Unit.

Processing unit	Initialization [s]	Calculation time (1 iteration) [s]	Deinitialization [s]
apl09 (0, 1)	0.0003686	0.4792336	0.0508268
apl10 (0, 1)	0.0003786	0.4921547	0.0553940
apl11 (0, 0)	0.0000857	0.1113934	0.2174853
apl11 (0, 1)	0.0000858	0.1115055	0.2900736
apl12 (0, 1)	0.0004564	0.5933427	1.3193764
apl12 (0, 2)	0.0004204	0.5465164	1.3072380

represents actual time (in seconds) needed for the calculations, Diff denotes the difference and Diff [%] denotes the difference as percentage of the time returned by the lp_solve tool. For detailed information including the configurations please see Appendix B, Tables B.1, B.2 and B.3 respectively.

As can be seen from the results the actual times needed to perform the calculations are similar to that returned by lp_solve. In most of the cases the relative difference does not exceed 5%. Only for cases with a very small number of equations the difference between the real and theoretical time exceeds 10%. For problems with a larger number of equations the difference falls into 0.41% - 5%. In all cases the configuration returned from lp_solve gave results faster than other tested configurations.

We also compared the results obtained using lp_solve with arbitrary selected and proportional assignments as well as setups generated using the min-min algorithm [30]. The comparison of real calculation times can be seen in Tables 4 and 5. For detailed information including the configurations please see Appendix B, Tables B.4 and B.5. The three arbitrary assignments (marked 1, 2 and 3) take into account relative performances of Processing Units (unit $PU_j^{N_i}$ with expected $P_{PU_j^{N_i}} > P_{PU_l^{N_k}}$ is assigned a larger number of packets than unit $PU_l^{N_k}$, based on preliminary knowledge on the devices) but without consideration of precise proportions. These show that there are very significant differences between total execution times for unoptimized and optimized configurations. This, in turn, justifies the approach that engages a more complex algorithm for bringing significant benefits over arbitrary configurations. The configuration in the column marked as proportional assigns data proportionally to the Processing Unit performance (with consideration of memory capacity as discussed in Section 4.5.2 – the excessive equations were assigned proportionally across other nodes). This approach does not take into account transfer times between nodes and from the node to the Processing Units thus giving worse results than from the proposed model. The Min-min v1 column shows results generated by the min-min algorithm taking into account single Jacobi equation calculation time and data transfer time between nodes. The last column, denoted Min-min v2, takes into account single Jacobi equation calculation time, data transfer time between nodes and the initialization and deinitialization times (as detailed in Section 4.5.2).

In the presented tests lp_solve always provided configurations with, usually significantly, better run times than the others. Furthermore the times returned by the min-min algorithm were distant from real calculation times, e.g. for 2048 Jacobi equations of size 2048 the time returned by the min-min algorithm, when the computation, transfer and initialization/deinitialization times were taken into account, was 768.318 seconds, whereas the real computation time was 1404.234 seconds. We

TABLE 8. Profiling results (in seconds) for problems of size 1024 with the use of CPU as Processing Unit.

Processing unit	Initialization [s]	Calculation time (1 iteration) [s]	Deinitialization [s]
apl09 (0, 1)	0.0005222	0.6788314	0.0747633
apl09 (1, 0)	0.0152677	19.8480542	0.2427339
apl10 (0, 1)	0.0005559	0.7226309	0.0905048
apl10 (1, 0)	0.0153396	19.9414649	0.2535768
apl11 (0, 0)	0.0001476	0.1919251	0.3242558
apl11 (0, 1)	0.0001517	0.1972112	0.3373213
apl11 (1, 0)	0.0060809	7.9051094	0.2780281
apl12 (0, 0)	0.0036818	4.7863524	0.8856008
apl12 (0, 1)	0.0006830	0.8878551	1.5552522
apl12 (0, 2)	0.0006246	0.8119639	1.6193448

thus believe that this makes our proposed approach a useful tool for deducing desired configurations in heterogeneous environments as it provides results very close to the real computation times and allows great detail in modeling of the whole calculation process. Its performance with timeouts is discussed further. It should be noted, that it would be possible to scale a solution with, e.g. X problems onto 2X, 3X, etc. numbers of problems by proportional assignment, however, without the guarantee of it being optimal. For instance, if we consider a system with 2 identical compute devices and 3 problems (2 problems assigned to 1 device and 1 to the other) then a proportionally scaled solution for 6 problems (4 problems to 1 device, 2 to the other) would not be optimal.

4.5. Discussion

As presented in the previous section the proposed model and usage of lp_solve produces viable configurations for data assignment in a heterogeneous environment. Some precautions need to be taken, however, as proper hardware profiling is the key to this solution.

4.5.1. *Hardware disparity*

In the paper we discuss a heterogeneous environment which, by nature, consists of nodes described by different values of parameters. During hardware profiling we, however, observed, that even similar hardware show some substantial differences. Such differences can be observed in Table A.2, where two servers (*apl09* and *apl10*) of the same model and connected to the same switch show different transfer times, and in Table A.4 where two identical GPUs, located within the same server (*apl11*) show different transfer times over PCI Express lanes. In the first case the differences can be caused by the cables and in the second case the system showed slightly slower transfer times over PCI express slots located further on the lane. Such differences have to be taken into account during optimal solution calculation.

4.5.2. *Computation time and memory usage*

First of all, care needs to be taken when scheduling data onto compute nodes as total calculation times are memory dependent. If the data scheduled fits into the RAM the times remain constant for such a configuration. After the data volume reaching 100% of RAM the computation times remain the same, however initialization and deinitialization times rise by a level of magnitude. Sample initialization and deinitialization times for Processing Units of *apl09* and *apl10* servers when calculating problems of size 2048 are presented in Table 6. In our approach, especially during tests with problems of size 2048, we introduced a limit of how many equations can be sent to given compute node. The limits are as follows: *apl09* and *apl10* – 348 equations, *apl11* – 1000 equations, *apl12* – 2000 equations. This limit is hardware related and should be adjusted on an individual basis. This can be seen as a practical way of dealing with these memory constraint effects. Given these constraints, this results in smaller ILP solver running times. For problems of size 512 and 1024 we did not introduce a limit as they are considerably smaller in terms of memory consumption.

4.5.3. *CPU as a Processing Unit*

Thanks to usage of OpenCL as a language for computational kernels in evaluation of our proposition we can run the code on almost any devices. Care need to be taken with such an approach as utilization of all physical cores of a CPU can actually hinder performance. When all physical cores are allocated to the calculation process the system needs more time to perform system calls thus making the initialization, deinitialization and computation times per one Jacobi solver iteration, for calls made to the GPUs, higher. An example of such behavior for problems of size 1024 can be seen in Tables 7 and 8. The impact of initialization and deinitialization times depends on the algorithm. In our case the initialization and deinitialization is done for every problem and total computation time greatly depends on the number of iterations used in Jacobi approximation. In our tests we run the approximation for 1300 iterations thus every increase of the time needed to perform 1 iteration impacts the final computation time to a great extent.

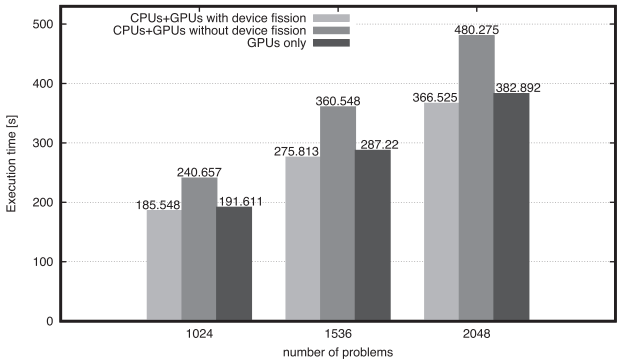

FIGURE 2. Comparison of optimal values for CPUs+GPUs with device fission, CPUs+GPUs without device fission and GPUs only for problems of size 1024.

As a result, using only GPUs for calculation yields better results than using CPUs and GPUs in the default configuration. An example of such a situation for problems of size 1024 can be seen in Fig. 2. The aforementioned increase of initialization and deinitialization times are to blame for this situation. In such an approach, the lp_solve tool could not take this increase into account and recommended not to use CPUs at all. The solution to this problem turned out to be usage of the *device fission* concept as described in Section 4.3. This way we can utilize the CPUs without impacting the performance. All other results in the paper are reported for this best configuration with device fission. It should be noted however, that the CPU is usually much slower than the GPU. It some cases, especially when there is a low number of CPUs available it might be beneficial to focus on GPUs only in terms of coding time.

4.5.4 Integer linear programming calculation times

The results presented in the previous sections of this paper show that an integer linear programming solver can be used to find optimal solution for data assignment. This applies to problems of limited size unfortunately, as integer programming has a high complexity and computation time. With many variables the time needed to find the optimal solution can exceed the time to actually run the application with even a random configuration. The times needed to find the optimal solutions, in conjunction with actual application running times, for our testbed application are presented in Tables 9, 10 and 11.

The complexity of lp_solve increases greatly with the number of variables. As seen in the aforementioned tables the times needed to find the optimal solutions are very high, around 2–7 times higher on average, even up to 48 times higher (for the worst case – 2048 problems of size 512) than the actual computation time. Times to find the optimal solution for size 2048 are visibly smaller, apparently due to additional constraints imposed due to memory limitations, as discussed in Section 4.5.2.

We decided to limit the time for lp_solve to 30, 60 and 90 seconds. The lp_solve time and actual computation times

TABLE 9. lp_solve runtime, theoretical computation time and real computation time for returned configuration (in seconds) for problems of size 512.

Number of problems	lp_solve solver time to obtain optimal configuration [s]	Theoretical application execution time [s]	Real application execution time [s]
64	0.43	5.303	6.186
128	3.39	10.404	11.03
256	37.98	20.532	22.085
512	310.914	40.674	41.759
768	1014.695	60.747	63.449
1024	2402.737	81.041	82.04
1536	4829.618	121.287	123.238
2048	7844.227	161.718	167.277

TABLE 10. lp_solve runtime, theoretical computation time and real computation time for returned configuration (in seconds) for problems of size 1024.

Number of problems	lp_solve solver time to obtain optimal configuration [s]	Theoretical application execution time [s]	Real application execution time [s]
64	0.255	11.939	13.522
128	2.819	23.464	24.939
256	33.874	46.101	46.428
512	200.967	92.03	91.917
768	512.156	138.038	137.888
1024	891.536	183.982	185.548
1536	2335.38	275.75	275.813
2048	5049.07	367.474	366.525

TABLE 11. lp_solve runtime, theoretical computation time and real computation time for returned configuration (in seconds) for problems of size 2048.

Number of problems	lp_solve solver time to obtain optimal configuration [s]	Theoretical application execution time [s]	Real application execution time [s]
64	0.087	32.357	33.861
128	4.723	63.368	64.619
256	22.752	125.485	125.442
512	148.998	248.674	248.16
768	321.409	373.62	379.882
1024	658.16	496.441	494.486
1536	572.939	807.889	797.254
2048	62.797	1304.797	1303.96

TABLE 12. lp_solve runtime (with and without timeouts) and real computation times for returned configurations (in seconds) for problems of size 512.

Number of problems	lp_solve time without any timeout [s]	Application running (calculation) time for configuration without lp_solve timeout [s]	Total running time for configuration without lp_solve timeout [s]	lp_solve time with timeout set [s]	Application running (calculation) time for configuration after lp_solve timeout [s]	Total running time for configuration after lp_solve timeout [s]	Total running time with lp_solve timeout to total running time without timeout [%]
256	37.98	22.085	60.065	30.018	29.606	59.624	99.266
	37.98	22.085	60.065	39.44	22.085	61.525	102.431
	37.98	22.085	60.065	38.594	22.085	60.679	101.022
512	310.914	41.759	352.673	30.018	69.297	99.315	28.161
	310.914	41.759	352.673	60.017	60.17	120.187	34.079
	310.914	41.759	352.673	90.018	60.17	150.188	42.586
768	1014.695	63.449	1078.144	30.017	101.771	131.788	12.224
	1014.695	63.449	1078.144	60.017	98.154	158.171	14.671
	1014.695	63.449	1078.144	90.025	98.154	188.179	17.454
1024	2402.737	82.04	2484.777	30.018	138.091	168.109	6.766
	2402.737	82.04	2484.777	60.018	134.137	194.155	7.814
	2402.737	82.04	2484.777	90.059	136.068	226.127	9.1
1536	4829.618	123.238	4952.856	30.017	204.057	234.074	4.726
	4829.618	123.238	4952.856	60.018	204.057	264.075	5.332
	4829.618	123.238	4952.856	90.066	204.057	294.123	5.938
2048	7844.227	167.277	8011.504	30.017	284.388	314.405	3.924
	7844.227	167.277	8011.504	60.026	284.388	344.414	4.299
	7844.227	167.277	8011.504	90.017	284.388	374.405	4.673

for 256, 512, 768, 1024, 1536 and 2048 problems of sizes 512, 1024 and 2048 are presented in Tables 12, 13 and 14 respectively.

While looking at ratios of lp_solve times (with timeout set) to application running times in Tables 12, 13 and 14, we can see that we have covered a large spectrum of ratios ranging

TABLE 13. lp_solve runtime (with and without timeouts), theoretical computation times and real computation times for returned configurations (in seconds) for problems of size 1024.

Number of problems	lp_solve time without any timeout [s]	Application running (calculation) time for configuration without lp_solve timeout [s]	Total running time for configuration without lp_solve timeout [s]	lp_solve time with timeout set [s]	Application running (calculation) time for configuration after lp_solve timeout [s]	Total running time for configuration after lp_solve timeout [s]	Total running time with lp_solve timeout to total running time without timeout [%]
256	33.874	46.428	80.302	30.018	65.229	95.247	118.611
	33.874	46.428	80.302	34.897	46.428	81.325	101.274
	33.874	46.428	80.302	34.959	46.428	81.387	101.351
512	200.967	91.917	292.884	30.018	185.62	215.638	73.626
	200.967	91.917	292.884	60.018	160.312	220.33	75.228
	200.967	91.917	292.884	90.017	160.312	250.329	85.47
768	512.156	137.888	650.044	30.018	278.231	308.249	47.42
	512.156	137.888	650.044	60.017	252.633	312.65	48.097
	512.156	137.888	650.044	90.017	252.633	342.65	52.712
1024	891.536	185.548	1077.084	30.018	372.326	402.344	37.355
	891.536	185.548	1077.084	60.017	340.614	400.631	37.196
	891.536	185.548	1077.084	90.018	340.614	430.632	39.981
1536	2335.38	275.813	2611.193	30.018	558.942	588.96	22.555
	2335.38	275.813	2611.193	60.018	558.347	618.365	23.681
	2335.38	275.813	2611.193	90.018	558.347	648.365	24.83
2048	5049.07	366.525	5415.595	30.018	773.552	803.57	14.838
	5049.07	366.525	5415.595	60.018	773.552	833.57	15.392
	5049.07	366.525	5415.595	90.019	773.552	863.571	15.946

from: 0.105-1.747, 0.039-0.753 and 0.022-0.229 respectively. We can compare these ratios to other works such as [7] for selection of services for workflow scheduling (with potential rescheduling at runtime) in which ratios for one run of a solver (various algorithms such as ILP, GA and GAIN tested) are in the range of 0.007 (the smallest workflow for which

TABLE 14. lp_solve runtime (with and without timeouts), theoretical computation times and real computation times for returned configurations (in seconds) for problems of size 2048.

Number of problems	lp_solve time without any timeout [s]	Application running (calculation) time for configuration without lp_solve timeout [s]	Total running time for configuration without lp_solve timeout [s]	lp_solve time with timeout set [s]	Application running (calculation) time for configuration after lp_solve timeout [s]	Total running time for configuration after lp_solve timeout [s]	Total running time with lp_solve timeout to total running time without timeout [%]
256	22.752	125.442	148.194	22.597	125.442	148.039	99.895
	22.752	125.442	148.194	22.482	125.442	147.924	99.818
	22.752	125.442	148.194	22.441	125.442	147.883	99.79
512	148.998	248.16	397.158	30.018	491.252	521.27	131.25
	148.998	248.16	397.158	60.019	491.252	551.271	138.804
	148.998	248.16	397.158	90.018	393.862	483.88	121.836
768	321.409	379.882	701.291	30.019	803.186	833.205	118.81
	321.409	379.882	701.291	60.019	728.936	788.955	112.5
	321.409	379.882	701.291	90.018	728.936	818.954	116.778
1024	658.16	494.486	1152.646	30.017	1068.15	1098.167	95.274
	658.16	494.486	1152.646	60.019	1057.357	1117.376	96.94
	658.16	494.486	1152.646	90.019	1057.357	1147.376	99.543
1536	572.939	797.254	1370.193	30.018	1298.564	1328.582	96.963
	572.939	797.254	1370.193	60.018	1197.18	1257.198	91.753
	572.939	797.254	1370.193	90.018	1229.935	1319.953	96.333
2048	62.797	1303.96	1366.757	30.019	1363.669	1393.688	101.97
	62.797	1303.96	1366.757	60.019	1283.449	1343.468	98.296
	62.797	1303.96	1366.757	65.482	1264.589	1330.071	97.316

an optimal solution could not be found in reasonable time) to 0.104.

Our research shows, that the lp_solve tends to stick to a local minimum for a long time. For that reason increasing the timeout value does not generally yield benefits, especially for bigger numbers of problems, for problems of size 512 and 1024. However, the initial timeout of 30 seconds allows lp_solve to find a feasible solution and corresponding

computation time combined with the lp_solve time is usually far lower than the total time needed to find and calculate the optimal configuration. Only for problems of size 2048 both times are comparable, still the solution obtained with lp_solve's timeout takes less time, in most cases. For problem sizes of 512 and 1024 and larger numbers of problems (1024+) the total time needed to find a suboptimal solution and perform the calculations is as low as 5-40% of the total time needed to find and calculate the optimal solution and is usually much smaller than time needed to calculate the solution using random or proportional assignment (see Tables 4 and 5). Only in one of the tested cases the total suboptimal solution lp_solve time with timeout equal 30 seconds took 266 seconds longer than running the calculations using proportional assignment. In all other cases the total lp_solve and calculation time was considerably shorter.

5. SUMMARY AND FUTURE WORK

In the paper we have shown that linear programming can be used to find good data assignment for calculation problems in heterogeneous environment. The process of finding optimal configurations, however, is not always time-feasible as it can take a long time due to a large number of unknowns. This can be, however, mitigated by setting a timeout on lp_solve thus providing sub-optimal, yet still time-feasible data assignment that results in much better than random or device performance proportional approaches to packet assignment to Processing Units. The proposed model is also useful in multi-layer heterogeneous environments. The conclusions obtained within the research can be summarized as follows – we have shown that:

1. Theoretical times from our model are accurate compared to real results – for larger data sizes difference does not exceed 5%, with up to 16.7% for small data sizes.
2. OpenCL 1.2's device fission allowing sub-dividing of a device into two or more sub-devices is a feature that enables to obtain better performance in heterogeneous CPU+GPU environments compared to the default CPU+GPU or GPU-only configurations.
3. Using an integer linear programming solver (lp_solve) with a timeout allows to obtain significantly better total (solver+application) run times than runs without timeouts, also significantly better than arbitrary chosen configurations. Furthermore, we can see that testing timeouts of 30, 60 and 90 seconds, for problems of size 512, 1024 generally best total times were obtained for timeout 30 seconds while for size 2048 best results were typically for 60 or even 90 seconds.
4. There may be slightly different hardware performances for identically specified nodes or compute devices connected to various switch or PCI slots which suggests the need for precise profiling.

The model presented in the paper does not take into account calculations and data transfer overlapping within a single node and between the manager and worker nodes. Some thought can also be given to the order of the nodes in which the data is sent to. The proposed solution can be further extended to take into account power consumption. We can formulate equations that correspond to energy consumption during computations on CPU and GPU thus extending the model to take into account all required parameters. This, however, complicates the model and requires further research. We consider using the aforementioned MERPSYS simulation environment [11] to aid within this area.

Data availability

Detailed data is available in Appendices A and B, source data will be shared on request to the corresponding author.

Disclosure

The authors declare that there is no conflict of interest.

REFERENCES

[1] Augonnet, C., Thibault, S., Namyst, R. and Wacrenier, P.-A. (2011) Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput.*, 23, 187–198.

[2] Bleuse, R., Hunold, S., Kedad-Sidhoum, S., Monna, F., Mounié, G. and Trystram, D. (Sep. 2017) Scheduling independent moldable tasks on multi-cores with gpus. *IEEE Trans. Parallel Distrib. Syst.*, 28, 2689–2702.

[3] Cavicchioli, R., Capodieci, N., Solieri, M. and Bertogna, M. (2019) Novel Methodologies for Predictable CPU-To-GPU Command Offloading. In Quinton, S. (ed) *31st Euromicro Conference on Real-Time Systems (ECRTS 2019), vol. 133 of Leibniz Int. Proc. in Informatics (LIPIcs)*, pp. 22:1–22:22, Dagstuhl, Germany Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[4] Chandar, D.D., Sitaraman, J. and Mavriplis, D.J. (June 2013) A hybrid multi-gpu/cpu computational framework for rotorcraft flows on unstructured overset grids. In *21st AIAA Computational Fluid Dynamics Conf*. American Institute of Aeronautics and Astronautics, Inc, San Diego, CA, USA.

[5] Cheng, J., Wu, P. and Chu, F. (September 2019) Mixed-integer programming for unrelated parallel machines scheduling problem considering electricity cost and makespan penalty cost. In *2019 Int. Conf. on Industrial Engineering and Systems Management (IESM)*, pp. 1–5. IEEE, Shanghai, China.

[6] Ciznicki, M., Kurowski, K. and Weglarz, J. (September 2017) Energy aware scheduling model and online heuristics for stencil codes on heterogeneous computing architectures. *Cluster Comput.*, 20, 2535–2549.

[7] Czarnul, P. (2014) Comparison of selected algorithms for scheduling workflow applications with dynamically changing service availability. *J. Zhejiang Univ. Sci. C*, 15, 401–422.

[8] Czarnul, P. (2018) *Parallel Programming for Modern High Performance Computing Systems*. Chapman and Hall/CRC Press ISBN 9781138305953.
[9] Czarnul, P. (2018) Parallelization of large vector similarity computations in a hybrid CPU+GPU environment. *J. Supercomput.*, 74, 768–786.
[10] Czarnul, P. (2020) Investigation of parallel data processing using hybrid high performance cpu+gpu systems and cuda streams. *Comput. Inform.*, 3, in press.
[11] Czarnul, P., Kuchta, J., Matuszek, M.R., Proficz, J., Rosciszewski, P., Wójcik, M. and Szymanski, J. (2017) MERPSYS: an environment for simulation of parallel application execution on large scale HPC systems. *Simul. Model. Pract. Theory*, 77, 124–140.
[12] Czarnul, P., Proficz, J. and Drypczewski, K. (2019) Survey of methodologies, approaches, and challenges in parallel programming using high-performance computing systems. *Sci. Program.*, 2020.
[13] Czarnul, P., Proficz, J. and Krzywaniak, A. (2019) Energy-aware high-performance computing: Survey of state-of-the-art tools, techniques, and environments. *Sci. Program.*, 2019, 8348791:1–8348791:8348791.
[14] Czarnul, P. and Rościszewski, P. (January 2014) 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) *Distributed Computing and Networking Conference (ICDCN)*, pp. 66–80. Springer Berlin Heidelberg, Coimbatore, India.
[15] Ernstsson, A., Lu, L. and Kessler, C. (Feb 2018) Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *Int. J. Parallel Program.*, 46, 62–80.
[16] Frâncu, M. and Moldoveanu, F. (2014) An Improved Jacobi Solver for Particle Simulation. In Bender, J., Duriez, C., Jaillet, F., Zachmann, G. (eds) *Workshop on Virtual Reality Interaction and Physical Simulation*, pp. 125–134. The Eurographics Association, Bremen, Germany.
[17] Gajger, T. and Czarnul, P. (2018) Modelling and simulation of GPU processing in the MERPSYS environment. *Scalable Comput.*, 19, 401–422.
[18] Goossens, B., De Vylder, J. and Philips, W. (October 2014) Quasar: a new heterogeneous programming framework for image and video processing algorithms on cpu and gpu. In *2014 IEEE Int. Conf. on Image Processing (ICIP)*, pp. 2183–2185. IEEE, Paris.
[19] Goossens, B., Luong, H., Aelterman, J. and Philips, W. (March 2018) Quasar, a high-level programming language and development environment for designing smart vision systems on embedded platforms. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1316–1321. IEEE, Dresden, Germany.
[20] Blake A. Hechtman, Andrew D. Hilton, and Daniel J. Sorin. TREES: A CPU/GPU task-parallel runtime with explicit epoch synchronization. *CoRR*, abs/1608.00571, 2016.
[21] Kumar, M. and Vardhan, M. (2018) Data confidentiality and integrity preserving outsourcing algorithm for system of linear equation to a malicious cloud server. In *Big Data Management and the Internet of Things for Improved Health Systems*, pp. 24–51. IGI Global.
[22] Lee, C.-L., Lin, Y.-S. and Chen, Y.-C. (10 2015) A hybrid cpu/gpu pattern-matching algorithm for deep packet inspection. *PLoS One*, 10, 1–22.
[23] Lin, C.-S., Hsieh, C.-W., Chang, H.-Y. and Hsiung, P.-A. (2014) Efficient workload balancing on heterogeneous gpus using mixed-integer non-linear programming. *J. Appl. Res. Technol.*, 12, 1176–1186.
[24] Lustig, D. and Martonosi, M. (Feb 2013) Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *2013 IEEE 19th Int. Symposium on High Performance Computer Architecture (HPCA)*, pp. 354–365. IEEE, Shenzhen, China.
[25] Mittal, S. and Vetter, J.S. (July 2015) A survey of cpu-gpu heterogeneous computing techniques. *ACM Comput. Surv.*, 47.
[26] Ünlü, Y. and Mason, S.J. (2010) Evaluation of mixed integer programming formulations for non-preemptive parallel machine scheduling problems. *Comput. Ind. Eng.*, 58, 785–800.
[27] Öhberg, T., Ernstsson, A. and Kessler, C. (Mar 2019) Hybrid cpu–gpu execution support in the skeleton programming framework skepu. *J. Supercomput.*, 76, 5038–5056.
[28] Pereira, A.D., da Silva Ramos, L.E. and Góes, L.F.W. (2015) Pskel: A stencil programming framework for cpu-gpu systems. *Concurrency and Computation: Practice and Experience*, 27, 4938–4953.
[29] Rauhe, H., Dees, J., Sattler, K.-U. and Faerber, F. (September 2013) Multi-level parallel query execution framework for cpu and gpu. In Catania, B., Guerrini, G., Pokorný, J. (eds) *Advances in Databases and Information Systems*, pp. 330–343. Springer Berlin Heidelberg, Genoa, Italy.
[30] Rehman, S., Javaid, N., Rasheed, S., Hassan, K., Zafar, F. and Naeem, M. (2019) Min-min scheduling algorithm for efficient resource distribution using cloud and fog in smart buildings. In Barolli, L., Leu, F.-Y., Enokido, T., Chen, H.-C. (eds) *Advances on Broadband and Wireless Computing, Communication and Applications*, pp. 15–27. Springer International Publishing, Cham.
[31] Rościszewski, P., Czarnul, P., Lewandowski, R. and Schally-Kacprzak, M. (2016) Kernelhive: a new workflow-based framework for multilevel high performance computing using clusters and workstations with cpus and gpus. *Concurrency and Computation: Practice and Experience*, 28, 2586–2607.
[32] Rosciszewski, P. (2018) Optimization of hybrid parallel application execution in heterogeneous high performance computing systems considering execution time and power consumption. *CoRR*, abs/1809.07611.
[33] Rossi, R.A. and Zhou, R. (2016) Hybrid CPU-GPU framework for network motifs. *CoRR*, abs/1608.05138.
[34] Sandokji, S. and Eassa, F. (2018) Task scheduling frameworks for heterogeneous computing toward exascale. *International Journal of Advanced Computer Science and Applications*, 9.
[35] Soner, S. and Özturan, C. (2015) Integer programming based heterogeneous cpu-gpu cluster schedulers for slurm resource manager. *Journal of Computer and System Sciences*, 81, 38–56.
[36] Sourouri, M., Baden, S.B. and Cai, X. (10 2016) Panda: A compiler framework for concurrent cpu + gpu execution of 3d stencil computations on gpu-accelerated supercomputers. *International Journal of Parallel Programming*, 45.

[37] Teodoro, G., Kurc, T., Andrade, G., Kong, J., Ferreira, R. and Saltz, J. (January 2017) Application performance analysis and efficient execution on systems with multi-core cpus, gpus and mics. *Int. J. High Perform. Comput. Appl*, 31, 32–51.

[38] S Terence. *Opencl device fission for cpu performance*, 2014.

[39] Ubal, R., Jang, B., Mistry, P., Schaa, D. and Kaeli, D. (2012) Multi2sim: A simulation framework for cpu-gpu computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, PACT'12*, pp. 335–344. Association for Computing Machinery, New York, NY, USA.

[40] Wen, Y., Wang, Z. and O'Boyle, M.F.P. (December 2014) Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, pp. 1–10. IEEE, Goa, India.

Appendix A. Detailed profiling results

TABLE A.1 Average time (in milliseconds) needed for $PU_j^{N_i}$ to calculate 1 iteration of Jacobi method for given problem size

Processing Unit	Problem size		
	512	1024	2048
apl09(0, 1)	0.1151	0.4140	1.5546
apl09(1, 0)	5.0781	20.0501	67.6384
apl10(0, 1)	0.1162	0.4140	1.5539
apl10(1, 0)	4.3065	20.0363	78.9142
apl11(0, 0)	0.0462	0.0964	0.2619
apl11(0, 1)	0.0451	0.0933	0.2608
apl11(1, 0)	1.6276	6.3978	25.1866
apl12(0, 0)	0.9917	3.6538	17.6083
apl12(0, 1)	0.3488	0.5697	0.884
apl12(0, 2)	0.3534	0.5676	0.8631

TABLE A.2 Two-way transfer time ("ping-pong") between *apl12* and compute nodes (in milliseconds)

N_i	Packet size						
	1	512KiB	1MiB	2MiB	8MiB	16MiB	32MiB
apl09	0.099	10.858	20.184	38.721	151.064	300.724	1047.023
apl10	0.048	10.505	18.326	36.227	143.374	285.649	570.863
apl11	0.068	9.38	18.246	36.175	151.187	285.692	570.819
apl12	0.001	0.152	0.56	1.369	6.11	8.188	28.05

TABLE A.3 The values of a_{N_i} and b_{N_i} for compute nodes

N_i	a_{N_i}	Packet size					b_{N_i}
		1MiB	2MiB	8MiB	16MiB	32MiB	
apl09	0.00004929	112445513	113134106	112003566	112102776	44961074	98929407
apl10	0.00002414	134080755	117147582	117436034	117921680	117646286	120846467
apl11	0.00003422	118266729	116975475	109405047	124732869	117682182	117412460
apl12	0.00000056	2564006259	2593139127	2654231929	8072365027	1689374057	3514623280

TABLE A.4 Two-way transfer time ("ping-pong") between the compute node N_i and its Processing Units (in milliseconds)

$PU^j_{N_i}$	Packet size						
	1	512KiB	1MiB	2MiB	8MiB	16MiB	32MiB
apl09 (0, 1)	0.004	0.176	0.345	0.694	3.066	6.139	12.504
apl09 (1, 0)	0.004	0.345	0.729	1.327	3.03	5.697	11.95
apl10 (0, 1)	0.004	0.176	0.345	0.695	3.073	6.077	12.471
apl10 (1, 0)	0.004	0.353	0.716	1.424	2.93	5.961	12.306
apl11 (0, 0)	0.004	0.172	0.338	0.859	8.463	14	29.131
apl11 (0, 1)	0.004	0.171	0.336	0.905	6.264	14.55	31.075
apl11 (1, 0)	0.005	0.581	1.219	2.645	9.527	17.504	39.713
apl12 (0, 0)	0.005	0.562	1.143	2.158	11.348	23.22	34.723
apl12 (0, 1)	0.592	1.396	1.815	4.505	17.73	30.087	77.42
apl12 (0, 2)	1.059	1.744	2.092	5.35	21.024	42.573	80.208

TABLE A.5 The values of $a_{PU_j^{N_i}}$ and $b_{PU_j^{N_i}}$ for Processing Units

$PU^j_{N_i}$	$a_{PU_j^{N_i}}$	Packet size					$b_{PU_j^{N_i}}$
		1MiB	2MiB	8MiB	16MiB	32MiB	
apl09 (0, 1)	0.00000183	6198466604	6012080625	5304632553	5458829592	5272045118	5649210898
apl09 (1, 0)	0.00000207	2732787423	3507324374	7388613364	6290068253	5365832175	5056925118
apl10 (0, 1)	0.00000184	6199199512	5997363296	5290845927	5584441919	5247767803	5663923691
apl10 (1, 0)	0.00000209	2885840727	2961196592	8358067620	5535166179	5288129866	5005680197
apl11 (0, 0)	0.00000215	6312843915	4021130014	1654797578	3030355181	2217472179	3447319773
apl11 (0, 1)	0.00000222	6358281538	3686302082	2347839424	2024672275	2030550821	3289529228
apl11 (1, 0)	0.00000234	1643793698	1470863994	1828337349	2103328160	1510862680	1711437176
apl12 (0, 0)	0.00000234	1802519390	2067840362	1369128594	1413113664	2917260467	1913972495
apl12 (0, 1)	0.00029578	2504426913	779725773	951388853	1357756394	708900065	1260439600
apl12 (0, 2)	0.00052974	3012275173	643808268	802780525	778551730	891586151	1225800369

Appendix B. Configuration details

TABLE B.1 The configurations, expected time and average real time (from 100 runs) for problems of size 512

Number of problems	64	128	256	512	768	1024	1536	2048
Time [s]	5.303	10.404	20.532	40.674	60.747	81.041	121.287	161.718
Real Time [s]	6.186	11.03	22.085	41.759	63.449	82.04	123.238	167.277
Diff [s]	0.883	0.626	1.553	1.085	2.702	0.999	1.95	5.559
Diff [%]	16.65	6.018	7.562	2.667	4.449	1.233	1.608	3.438
d_{apl09}	19	38	75	148	221	296	443	591
d_{apl10}	18	36	72	145	217	288	433	577
d_{apl11}	23	46	92	183	274	366	549	732
d_{apl12}	4	8	17	36	56	74	111	148
$d^{apl09}_{(0,1)}$	19	37	73	144	215	287	430	573
$d^{apl09}_{(1,0)}$	0	1	2	4	6	9	13	18
$d^{apl10}_{(0,1)}$	18	35	70	140	209	278	417	556
$d^{apl10}_{(1,0)}$	0	1	2	5	8	10	16	21
$d^{apl11}_{(0,0)}$	11	22	43	85	127	170	255	340
$d^{apl11}_{(0,1)}$	11	21	43	85	127	170	254	339
$d^{apl11}_{(1,0)}$	1	3	6	13	20	26	40	53
$d^{apl12}_{(0,0)}$	2	4	9	18	28	37	55	74
$d^{apl12}_{(0,1)}$	1	2	4	9	14	19	28	37
$d^{apl12}_{(0,2)}$	1	2	4	9	14	18	28	37

TABLE B.2 The configurations, expected time and average real time (from 100 runs) for problems of size 1024

Number of problems	64	128	256	512	768	1024	1536	2048
Time [s]	11.939	23.464	46.101	92.03	138.038	183.982	275.75	367.474
Real Time [s]	13.522	24.939	46.428	91.917	137.888	185.548	275.813	366.525
Diff [s]	1.583	1.475	0.327	-0.112	-0.15	1.566	0.063	-0.948
Diff [%]	13.256	6.287	0.71	-0.122	-0.108	0.851	0.023	-0.258
d_{apl09}	11	24	48	95	143	191	286	382
d_{apl10}	12	24	48	96	144	192	288	384
d_{apl11}	36	70	139	278	417	556	834	1111
d_{apl12}	5	10	21	43	64	85	128	171
$d^{apl09}_{(0,1)}$	11	24	47	93	140	187	280	374
$d^{apl09}_{(1,0)}$	0	0	1	2	3	4	6	8
$d^{apl10}_{(0,1)}$	12	24	47	94	141	188	282	376
$d^{apl10}_{(1,0)}$	0	0	1	2	3	4	6	8
$d^{apl11}_{(0,0)}$	18	34	68	136	203	271	407	542
$d^{apl11}_{(0,1)}$	18	35	68	136	204	272	407	543
$d^{apl11}_{(1,0)}$	0	1	3	6	10	13	20	26
$d^{apl12}_{(0,0)}$	1	2	5	11	16	22	33	44
$d^{apl12}_{(0,1)}$	2	4	8	16	24	32	48	64
$d^{apl12}_{(0,2)}$	2	4	8	16	24	31	47	63

TABLE B.3 The configurations, expected time and average real time (from 100 runs) for problems of size 2048

Number of problems	64	128	256	512	768	1024	1536	2048
Time [s]	32.357	63.368	125.485	248.674	373.62	496.441	807.889	1304.797
Real Time [s]	33.861	64.619	125.442	248.16	379.882	494.486	797.254	1303.96
Diff [s]	1.504	1.251	-0.043	-0.514	6.262	-1.954	-10.636	-0.836
Diff [%]	4.647	1.974	-0.034	-0.207	1.676	-0.394	-1.316	-0.064
d_{apl09}	2	10	21	42	62	84	153	301
d_{apl10}	5	10	21	42	64	85	155	303
d_{apl11}	49	93	184	365	549	729	1000	1000
d_{apl12}	8	15	30	63	93	126	228	444
$d^{apl09}_{(0,1)}$	2	10	21	41	62	82	149	292
$d^{apl09}_{(1,0)}$	0	0	0	1	0	2	4	9
$d^{apl10}_{(0,1)}$	5	10	21	41	63	83	151	295
$d^{apl10}_{(1,0)}$	0	0	0	1	1	2	4	8
$d^{apl11}_{(0,0)}$	25	47	92	181	273	362	497	500
$d^{apl11}_{(0,1)}$	24	46	91	181	271	360	494	500
$d^{apl11}_{(1,0)}$	0	0	1	3	5	7	9	0
$d^{apl12}_{(0,0)}$	0	1	2	5	7	10	18	35
$d^{apl12}_{(0,1)}$	4	7	14	29	43	58	105	205
$d^{apl12}_{(0,2)}$	4	7	14	29	43	58	105	204

TABLE B.4 Comparison of calculation times between optimal, arbitrary and proportional to Processing Unit performance configurations for 2048 problems of size 1024

Configuration	Optimal	Arbitrary 1	Arbitrary 2	Arbitrary 3	Proportional	Min-min v1	Min-min v2
Real Time [s]	366.525	2680.943	1764.693	2598.601	537.559	726.168	395.024
d_{apl09}	382	312	465	365	168	209	412
d_{apl10}	384	312	465	365	168	216	425
d_{apl11}	1111	712	815	1100	1453	1239	1021
d_{apl12}	171	712	303	218	259	384	190
$d^{apl09}_{(0.1)}$	374	212	400	251	165	204	401
$d^{apl09}_{(1.0)}$	8	100	65	114	3	5	11
$d^{apl10}_{(0.1)}$	376	212	400	251	165	211	414
$d^{apl10}_{(1.0)}$	8	100	65	114	3	5	11
$d^{apl11}_{(0.0)}$	542	306	365	450	710	608	494
$d^{apl11}_{(0.1)}$	543	306	365	450	732	616	493
$d^{apl11}_{(1.0)}$	26	100	85	200	11	15	34
$d^{apl12}_{(0.0)}$	44	100	73	51	19	27	56
$d^{apl12}_{(0.1)}$	64	306	115	85	120	178	67
$d^{apl12}_{(0.2)}$	63	306	115	82	120	179	67

TABLE B.5 Comparison of calculation times between optimal, arbitrary and proportional to Processing Unit performance configurations for 2048 problems of size 2048

Configuration	Optimal	Arbitrary 1	Arbitrary 2	Arbitrary 3	Proportional	Min-min v1	Min-min v2
Real Time [s]	1303.96	8946.678	6963.740	11996.284	1713.970	1899.181	1404.234
d_{apl09}	301	312	465	365	189	166	295
d_{apl10}	303	312	465	365	187	170	303
d_{apl11}	1000	712	815	1000	1000	1000	1000
d_{apl12}	444	712	303	318	672	712	450
$d^{apl09}_{(0.1)}$	292	212	400	251	184	162	284
$d^{apl09}_{(1.0)}$	9	100	65	114	5	4	11
$d^{apl10}_{(0.1)}$	295	212	400	251	184	167	294
$d^{apl10}_{(1.0)}$	8	100	65	114	3	3	9
$d^{apl11}_{(0.0)}$	1000	306	365	400	496	495	493
$d^{apl11}_{(0.1)}$	0	306	365	400	497	495	491
$d^{apl11}_{(1.0)}$	0	100	85	200	7	10	16
$d^{apl12}_{(0.0)}$	35	100	73	51	16	17	42
$d^{apl12}_{(0.1)}$	205	306	115	134	324	343	204
$d^{apl12}_{(0.2)}$	204	306	115	133	332	352	204