PSciLab: An Unified Distributed and Parallel Software Framework for Data Analysis, Simulation and Machine Learning - Design Practice, Software Architecture, and User Experience

Stefan Bosse

University of Bremen, Dept. Mathematics & Computer Science, Bremen, Germany

Abstract. A hybrid distributed-parallel cluster software framework for heterogeneous computer networks is introduced, which supports simulation, data analysis, and Machine Learning (ML) using widely available JavaScript Virtual Machines (VM) and Web browsers to perform the working load. This work addresses parallelism primarily on control-path level and partially on data-path level targeting different classes of numerical problems that can be either data-partitioned or replicated. composed from a set of interacting worker processes that can be easily parallelised or distributed, e.g., for large-scale multi-element simulation or ML. The suitability and scalability for static- and dynamic sized problems is experimentally investigated with respect to the proposed multi-process and communication architecture and the data management using customised SQL data bases with network access. The framework consists of a set of tools and libraries, mainly the WorkBook (processed by a Web Browser) and the WorkShell (processed by node.js). It can be shown that the proposed distributed-parallel multi-process approach with a dedicated set of inter-process communication methods (message- and shared-memory-based) scales efficiently with the problem size and the number of processes. Finally, it is shown that the JavaScript-based approach can be easily used for exploiting parallelism by a typical numerical programmer and data analyst not requiring any special knowledge about parallel and distributed systems and their interaction. There is a focus on VM processing.

Keywords. Distributed and Parallel Simulation, Distributed and Parallel Machine Learning, Parallel Computing, Hierarchical Clusters

1. Introduction

Any numerical computation can be composed from a set of interacting functions. One case is a linear sequence of functions. But often computational functions can be divided into sets of parallel function evaluations. There are two classes of partitioned numerical computation:

1. Strongly coupled with extensive data dependencies and communication interaction;
2. Loosely coupled with low degree of inter-function data dependencies and communication.

Statistical data analysis, simulation, and Machine Learning are typical examples of computational problems that can belong to the second class and can be easily processed by parallel and/or distributed computational processes. The focus of this work are second class problems.

A heterogeneous cluster-based parallel and distributed numerical and machine learning framework is introduced using widely available JavaScript Virtual Machines (VM), either part of a Web Browser (client-side) or part of a dedicated server-side engine (e.g., node.js). It features an easy and explicitly controlled way to compose parallel and distributed numerical computation by worker processes that can be created and processed on a wide range of platforms and accessed and controlled by a Web Browser (Laboratory in the Browser). Parallelisation of numerical tasks was investigated since more than 60 years. This work addresses practical aspects and explores the problem space suitable for parallelisation and distribution.

This work provides no new theoretical contributions to parallel and distributed systems. Instead, this work proposes an unified software framework for hybrid distributed-parallel computation with a rigorous experimental evaluation of the performance of parallel and distributed data processing systems by typical numerical use-cases and by using consumer hardware including smartphones and widely available software like Web Browsers. The paper identifies optimised software and architecture details, suitable metrics to assess computational nodes in advance, and the classes of problems that are suitable for our proposed and implemented multi-process and communication architecture with a focus on VM technologies.

Primarily, parallel and network-connected hierarchically organised distributed-parallel data processing systems are addressed that are using Web browsers (WorkBook software) and command-line shells using node.js (WorkShell software) deployed in heterogeneous environments. Code can be executed independently of the underlying host platform and the programs. Program code can be processed on both architecture classes (mostly) without code modification. Both architecture classes can be coupled via communication channels. The novel Parallel Scientific Laboratory (PSciLab) software framework bundles a set of tools and modules to create numerical multi-processing using common JavaScript Virtual Machines (VM). This framework is a conceptual successor of the PsiLAB framework using originally the OCaml VM [1]. PsiLAB was limited to single process execution and used native code libraries to perform computational intensive operations (like matrix BLAS or FFT packages). The PSciLab software framework do not rely (mostly) on any native numerical code library supporting Web browser processing, too.

The first main goal of this work is related to processing architectures and the investigation of the suitability of low-cost hardware (i.e., mainly consumer electronics with in-

tegrated Intel CPU and GPU architectures, smartphones, and embedded computers) with respect to distributed and parallel scaling and overall performance. There is a focus on hybrid distributed-parallel systems combining parallel and distributed computing coupled closely. Finally, the benefit of integrated low-cost GPUs are evaluated. The second main goal is related to the deployment of Virtual Machines (VM) with JavaScript programming in heterogeneous networks. JavaScript has the advantage that it is widely used and that JS can be executed on any machine. Using Googles V8 core engine (e.g., part of Chrome Web Browser and node.js) native code performance can be reached. The easy-to-learn and easy-to-use JS-based parallel data processing framework (e.g., distributed ML requires less than 500 lines of code) is the main advantage of this framework over other frameworks like Python or TensorFlow, for example. The usage of Web Browsers simplifies operation since no software installation is required. In contrast to Jupiter notebooks interfacing a separated Python VM instance, the WorkBook is self-contained and performs all computations.

Two use-cases addressing large-scale multi-entity simulation and multi-model ML are used to demonstrate the capabilities of the software architecture and programming environment.

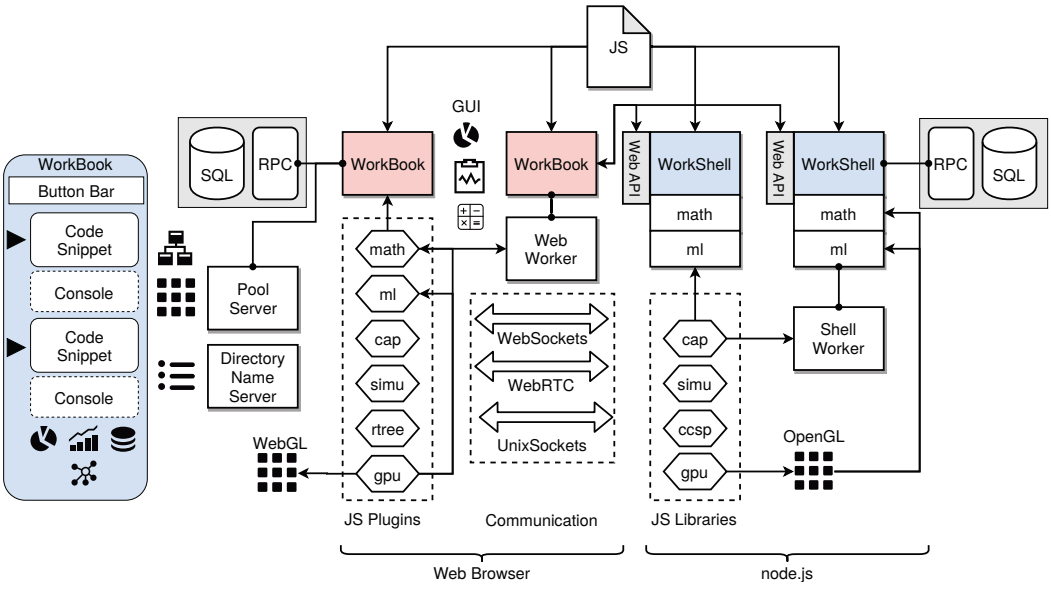


Fig. 1. PSciLab software framework and its components (Right) and WorkBook GUI layout (Left) using code snippets flows with micro consoles

An overview of the PSciLab software framework and its components is shown in Fig. 1. This software framework enables hybrid and hierarchical parallel and distributed

processing using the following components:

1. WorkBook: The main software components are the WorkBook processed by a Web Browser that provides a GUI including extensive graphical data visualisation like data plotting with code snippet work flows similar to, e.g., MatLab or the Jupiter Notebook. A WorkBook can be used stand-lone or together with WorkShells (or other WorkBooks). A WorkBook can spawn Web workers to perform computational tasks locally in parallel or shell workers for distributed computing. Scripts executed in workers (Web and Shell) started from a WorkBook can execute visualisation operations directly.
2. WorkShell: The second main component is a head-less and terminal-based WorkShell that is processed by node.js [2] and that can be accessed by WorkBooks directly. The WorkShell can be used stand-alone (executing scripts from command line) or together with WorkBooks provided by a Web Service API. A WorkShell can spawn system workers to perform computational tasks in parallel.
3. SQLite: The third main component is a set of (distributed) customised SQLite data bases with remote network access services via a JSON-based remote procedure call (RPC) interface used for input, intermediate, and output data exchange.
4. Pool/Proxy: An optional worker pool server and Internet proxy for connecting workers (WorkBooks and WorkShells with their worker processes) and to provide a group service.

The entire framework as well as the user code is programmed entirely in JavaScript ensuring high portability. Additionally, there is a Web WorkShell component providing a Web-based WorkShell (with a terminal console) for Web Browsers extended with a sub-set of the WorkBook and common libraries. The Web WorkShell (WWS) can be integrated in any Web page. The WWS can be configured by URL parameters including script loading. It is mainly used for participative crowd computing. Finally, interconnectivity of computational processes among the Internet is provided by a pool and proxy server via WebSocket communication.

The next sections introduce fundamentals of the multi-processing architecture, data management, the parallel and distributed programming of workers, and the software framework itself. Finally, an extended use-case section evaluates the benefits of the PSciLab processing framework for two prominent applications. It can be shown that the VM-based computation can compete with native approaches and there is a suitable scaling and speed-up that can be reached by worker groups.

The novelty of this framework is its high portability and the support of strong heterogeneous networks of computers, the Web Browser as a scientific laboratory, tight scaling to large-scale problems, the void of special software or hardware components, and the ease of programming without expert knowledge (regarding parallel and distributed systems). Hierarchical parallel-distributed clusters are supported seamlessly using an

unique API.

2. Related Work

The computational performance of modern computer systems developed fastly with a nearly exponential grow of operational speed and memory capacity. In the last decade there was a stagnation. The memory capacity doubles still about every year, but the operational throughput (computational powers) kept nearly constant. But the demand of computation doubles currently every three month, mostly driven by ML and data mining tasks. Parallel computation applied to numerical problems is a long existing field in computer science. Originally starting with networked computer clusters in the 1960's (clusters of workstations), the rise of dedicated parallel computers in the 1980's, and finally cloud computing using distributed server farms appearing as one virtual machine. With the rise of multi-core computers consumer software and common widely used numerical software like MatLab [3] try to exploit parallelism. Parallel computation is commonly used to reduce the computational time by vertical (sub-tasks applied to partitions of the input data) and horizontal (pipe-lines applied to different input data) parallel processing. There is control-path and data-path parallelism that has to be distinguished. Control-path parallelism can be easily created by multiple worker processes executed on different machines (clusters) or CPUs (cores). Data-path parallelism can only be achieved with dedicated hardware (although, CPUs support VLIW extensions) by Field-programmable Gate Arrays (FPGA) or customised ASICs, today mostly performing general purpose computing on Graphical Processing Units (GPGPU), but primarily being still fine-grained control-path parallelism, or by hybrid approaches [4].

Two different dominant parameters were identified in [3] that have a significant impact on the speed-up of parallel systems: (1) The memory model (shared memory vs. distributed memory, memory architecture, cache hierarchy, bandwidth etc.); (2) The granularity of the parallel tasks on data- and control-path level, moreover the parallel execution time compared with the overhead (by set-up and communication). E.g., the start-up time of a JS VM with the WorkShell code can consume up to one second (see Sec. 8.1 for details). The third important parameter concluding from (1) and (2) is the communication load and communication architecture. Communication introduces synchronisation, and synchronisation introduces temporal sequential processing that reduces the degree of parallelism that can be achieved.

With the rise of the Internet and the Web, high-bandwidth networks, and powerful Web browser software capable of executing JavaScript efficiently server-less computation using interconnected Web browser was successfully deployed for numerical computations [5]. The Web browser as a computational node is a central part in this work, too. The JS VM is the key methodology and technology for cluster computations in heterogeneous environments. JS is always passed to the VM in a architecture independent text format (UTF8 code) and can be easily exchanged. Moreover, any JS object and even

functions can be serialised to text by the JavaScript Object Notation (JSON) format and again deserialised to code or data objects. Beside core data types like numbers and strings, JSON supports only non-nested record data structures and polymorphic arrays. An extended version JSONfn is used in this work using a modified serialiser and deserialiser that encodes functions, typed arrays, matrix objects, and generic data buffers, too. By using Web Browsers, peer-to-peer communication can be realised by WebRTC [5] or by using a proxy server and WebSockets. WebSockets as an upgrade of a HTTP(S) connection that requires an external HTTP service, whereas WebRTC depends on external signalling (STUN) and relay (TURN) servers. WebRTC data channels enable direct browser-browser (and browser-server) communication. But symmetric Network Address Translation (NAT) prevents direct IP-UDP-based communication and requiring a relay server. For this reason and the non-availability of WebRTC data channels in Web workers, WebSockets (and generic HHTP requests) together with an external proxy server are used in this work only. Generic single HTTP requests (GET/PUT methods) are preferred for Remote Procedure Calls (RPC), whereas WebSocket channels are used for data and control streams, primarily.

Messaging and communication architectures were identified early as a central limiting factor with respect to achievable speed-up and scaling as a function of the problem size that has to be addressed by suitable communication models [6]. Algorithms and structures of parallel programs has to be adopted carefully to benefit from parallel processing substantially [7]. There is no universal parallel computer and communication architecture that matches any problem and algorithm efficiently. Parallel simulation of large-scale multi-entity systems (e.g., Multi-agent Systems or Cellular Automata) is one major use-case that is addressed by parallel and distributed computation [6,8]. Parallel Cellular Automata (PCA) used to compute or simulate complex systems are still under research, e.g., in materials science and micro-structure simulation [9] or for image processing [10]. Spatial domain partitioning is commonly applied to the cell grid to achieve a parallel decomposition of the computational problem. In [11] the load balancing and CPU allocation is identified as a key challenge in parallel (and distributed) computation of CA. In this work a pool server is introduced that is capable to manage computational groups and worker processes on demand (Computation as a Service architecture).

Beside micro-scale simulation, macro-scale simulation like urban simulation (traffic, migration, segregation, energy supply, etc.) or pandemic simulation with a high number of cells or entities are demanding for parallel and distributed computation due to high computational complexity [12] and complex long-range communication graphs. Agent-based simulation is another field for parallel computation [13]. In contrast to Cellular Automata (CA) with short-range entity-entity communication (data dependency) suitable for parallel decomposition with nearly linear scaling with respect to the problem size and number of processors, Multi-agent systems can be characterised by long-range communication, limiting the scaling and achievable speed-up significantly and limiting efficient distribution of the computation. But domain-specific partition and communication restrictions can relax this limitation. The first use-case evaluated in this paper ap-

plies spatial world partitioning for parallel processing of a CA. The partitioning scheme commonly relies on a shared or distributed memory model. Simulation including MAS can exploit data-path (shared memory) parallelism using GPGPU acceleration and control-path parallelism with network clusters equally [13]. Hardware acceleration of agent-based simulation using GPGPU and Accelerated Processing Units (APU) is still an ongoing research field [14].

The second major field today demanding for parallel computation is data-driven modelling using Machine Learning (ML) methods and high-dimensional data analysis. In contrast to multi-entity simulation with a spatial context enabling parallel decomposition on control-path level, ML requires commonly data-path parallelism with a high-bandwidth shared memory architecture to achieve a reasonable speed-up and scaling. But multi-model, ensemble, and distributed ML can benefit from control-path parallelism even using low-resource devices with low computational power and memory storage [15]. The second use-case evaluated in this work is multi-model training for parallel hyper-parameter space exploration. In [16] the authors discuss JavaScript-based Artificial Neural Network software frameworks and the suitability of Web browser for ML tasks. They conclude that Web browser can compete with native code server-based frameworks like TensorFlow.

There are only a few systems using large-scale artificial neural networks (Deep ANN) partitioned over several nodes, discussed in [17] and [18], and there are communication limitations preventing efficient control-path parallelism for the training of smaller network sizes (less than 100000 parameter variables). Most current systems do not split the ANN itself and each physical computing node handles the entire ANN [19]. Instead data-path parallelisation is done by GPU (e.g., CUDA API with the cuDNN library [20]) and FPGA accelerators. Cloud computing, e.g., using the Hadoop framework based on Map & Reduce methodologies, is increasingly used for large model sizes. The decomposition of large predictive model into a forest of smaller partial models (e.g., in a multi-class classification problem each model is a single class predictor, trained stochastically [17]) with model fusion is a suitable methodology for distributed training with low communication overhead.

Common parallel and distributed programming and communication environments are the Parallel Virtual Machine (PVM), recently extend to heterogeneous systems, including CPU/GPU hybrid systems [21], and the Message Passing Interface (MPI) [22]. Both frameworks require expert knowledge for the efficient design of distributed-parallel systems.

The architecture of data input-output streams as part of the communication infrastructure of parallel and distributed system has a significant impact of speed-up and scaling. Memory-in-place methodologies are only suitable for parallel systems using shared memory architectures. Distributed memory should be organised and the communication overhead must be considered carefully. Data bases can provide the necessary organisation of data. In this work, a set of relational SQL data bases is used. But SQL data bases do no scale linearly with increasing data volume and complexity. It is difficult to

distribute partial content of data bases. NoSQL data bases typically are chosen for large-scale cloud computing and data mining tasks [23]. Temporal data bases are suitable for the processing of latent temporal data stream and high amount of temporary data. The SQL data base was chosen here with its lightweight implementation SQLite that can be embedded efficiently in any application program, even in a Web Browser using transpilation technologies. SQLite provides an advanced cache algorithm that reduces file access significantly in distributed computations like multi-model ML (reading the same input data multiple times).

3. Problem Formalization and Taxonomy

The taxonomy of parallel and distributed applications is related to data- and application classes, e.g.,

- Data classes: Vector, Matrix, Tensor, Functional data (Cellular Automata);
- Algorithm classes: Matrix operations in general, data-driven and iterative optimisation problems, Cellular Automata processing, simulation, Equation Solving, Regression, Statistical Analysis;
- Data dependency classes: Local, global, clustered, static- and dynamic content, static- and dynamic sizes, and horizontal (time) and vertical dependencies;
- Processing flow classes: Data flow $\Leftrightarrow$ Functional flow, Control flow $\Rightarrow$ Synchronisation;
- Partitioning classes: Single-data and single-model versa multi-data and multi-model computation (e.g., ensemble model ML with model fusion, data streams);
- Model classes: Data-driven modelling, e.g., using ML methodologies, split in training-, test-, and validation-phases, hypothesis test and model selection (parallel model space exploration), hyper-parameter space exploration;
- Size classes: Static size versa dynamic size problems.

Data Classes

There is the following data class taxonomy addressed by a numerical software framework:

Data Dimension

Matrix (and vector) operations rely on multiply-add and loop instruction that require fine-grained parallelism on data-path level. If independent matrix segmentation is possible with respect to input and output data, then coarse-grained parallelism on control-path can be applied partially. A Cellular Automata with a grid world is an examples for the deployment of control-path parallelism.

Data Sets

If there is an unordered data set $\tilde{D}=\{d_\mathrm{i}\}_\mathrm{i}$ consisting of independent data items, then these data items can be processed independently and in parallel, i.e., $d_1 \rightarrow f^1 \parallel d_2 \rightarrow f^2 \parallel ..$, where f^j are replicated functions derived from a master f.

Data Streams

If there is an ordered sequential data stream $\tilde{D}=[d_\mathrm{t}]_\mathrm{t}$ consisting of independent data items provided in a sequential stream, and there is a functional chain $f_1 \rightarrow f_2 \rightarrow .. \rightarrow f_\mathrm{n}$, vertical control-path level parallelism can be exploited efficiently in a functional chain, i.e., $d_\mathrm{j} \rightarrow f_\mathrm{i} \parallel d_{\mathrm{j}+1} \rightarrow f_{\mathrm{i}-1} \parallel d_{\mathrm{j}+2} \rightarrow f_{\mathrm{i}-2}$ and so on. Functional pipelines can be used to exploit functional-level parallelism on horizontal control-path level.

Model Sets

If there is an non-unique and non-deterministic optimization problem using data sets, e.g., ML, i.e., $\tilde{M}=\{m_\mathrm{i}\}_\mathrm{i}$, then control-path level parallelism (multi-processing) can be used to derive the set of models $\tilde{M}$ from the same data set (or any sub-set) but with different parameter variable set (including random variables), i.e., $f^1(D,p_1) \rightarrow m_1 \parallel f^2(D,p_2) \rightarrow m_2 \parallel .. \parallel f^\mathrm{n}(D,p_\mathrm{n}) \rightarrow m_\mathrm{n}$, where f^j are replicated functions derived from a master f. .

Data sets can be distributed on different computers and can be classified into two different fragmentation classes [24]:

Horizontal Fragmentation

There are sub-sets of instances that are stored at different computing nodes $\tilde{N}=\{N_\mathrm{i}\}_\mathrm{i}$, i.e., $\tilde{D}=\{d_\mathrm{i}\}_\mathrm{i}$, $d_1 \rightarrow N_1$, $d_2 \rightarrow N_2$, .., $d_\mathrm{n} \rightarrow N_\mathrm{n}$.

Vertical Fragmentation

There are sub-sets of attributes (variables) of instances that are stored at different sites.

The data set fragmentation is an important feature that is exploited by the our parallel and distributed data processing framework.

Problem Classes

There are two different problem classes with respect to the computational size (number of intsructions) and data size (number of data elements):

Static

The problem size is static, i.e., $|D|=s=const$ and do not change with parallelisation, i.e., by increasing the number of processing nodes N_i and processes $P_\mathrm{j}(N_\mathrm{i})$. Static sized problems are well suited for partition like in grid-based simulation (CA), i.e., the full data set is partitioned and processed on multiple nodes (different data on different nodes): $D=\{d_\mathrm{i}\}_\mathrm{i}$, $d_1 \rightarrow f^1 \parallel d_2 \rightarrow f^2 \parallel .. \parallel d_2 \rightarrow f^\mathrm{k}$. The aim of parallelisa-

tion is the reduction of the overall computation time, ideally by $1/PN$ if PN is the number of parallel processes.

Dynamic

The problem size grows with increasing number of processing nodes and computational processes. An example is multi-model ML training. Each processing node trains an independent model with different parameter variables from the set $V=\{v_i\}_i$ including random variables (Monte Carlo simulation), either using the same input data D or different input data D_1, .., D_u, or sub-sets $d_i \in D$, i.e.: $D, v_1 \rightarrow f^1$, .., $D, v_n \rightarrow f^n$. The aim of parallelisation is ideally a constant computational time with increasing number of parallel processes (scaling is $S=1$), or at least a growth lower than $PN=|P|$, i.e. $1 < S < PN$.

The presented software framework addresses both problem size classes. Two use-cases will demonstrate each problem class.

Algorithmic Classes

In this work control-path parallelism is addressed primarily. This constraint requires the capability of algorithms to be partitioned into tasks with low (or medium) data inter-dependency and low inter-task communication (i.e., synchronisation). Although, shared memory communication is supported, the communication overhead should be kept low to achieve a nearly linear scaling with the number of processors (in the range 4-100). Some prominent examples satisfying this constraint are some Machine Learning applications and large-scale multi-entity simulation. If the input and output data can be partitioned into independent data sub-sets then this criteria can be met immediately.

Machine Learning

Data-driven Machine learning (ML) can profit from two parallelism methods:

Data-Path Parallelism

Applied to function evaluation and matrix operations

Control-Path Parallelism

Applied to training of multiple models (multi-instance ensemble learning and/or parameter space exploration) or applying to inference on multiple data sets and/or multiple model instances.

ML can be further classifies in a taxonomy similar to Flynn's computer architecture, but here using model instances for training (deduction) and inference (induction):

STSI

Single-model instance training with single-model instance inference (i.e., 1:1 mapping of input and output data to models); input and output data is not partitioned

MTSI

Multi-model instance training with single-model instance inference, either by model and parameter space exploration selecting the best model or by model fusion; input and output data is not partitioned

MTMI

Multi-model instance training with multi-model instance inference, i.e., primarily distributed ML; input and output data is partitioned and single model instances operate only on a (local) sub-set of the input data; final global model fusion is optional;

STMI

Single-model instance training with with multi-model instance inference, e.g., pixel-based feature detection applied to images; replication of the trained model; input data for training is not partitioned, input data for inference data can be partitioned

ML is typically a minimisation problem iterating over a set of data instances repeatedly. A single ML task itself has no inter-process data dependencies during the training and inference phases. ML can therefore profit from pure control-path parallelism and distribution, but the parallelisation of one training task poses high inter-layer data dependency and communication and therefore can only profit from data-path parallelism (e.g., used by Tensorflow with GPU back-end) applied to matrix operations. The input training and test data need only be transferred one time and the communication time can be commonly neglected (compared with the training task). The iterative optimisation process that minimises an error/loss function by parameter adaptation uses the same input data repeatedly. Commonly there is a forward pass that computes the model with given input data finally applying the loss function to compute the error, followed by a backward pass that adjusts the model parameters based on the error. Artificial Neural Networks (ANN) and function regression are prominent models that are trained by forward-backward propagation algorithms. This allows efficient parallel computation on the same node (including outsourced GPGPU computations) as well as distributed computation on different independent computing nodes for multi-model training.

Simulation

Large-scale multi-entity simulation is an computational intensive task. An entity is either passive like a volume in Finite Element Methods (FEM) used for physical and mechanical simulation, or active either as a simple state-based turing machine (a state-based Finite State Machine, FSM) in a cell of a Cellular Automata (CA) or a more complex agent in a Multi-agent System (MAS). FEM and CA cells pose short-range interaction leading to static short-range (local) data dependencies (and data exchange), whereas agents can pose short- and long-range interaction up to global data dependencies and global communication. I.e., FEM/CA cells have a fixed set of neighbouring elements, whereas agents have dynamic sets of interacting agents. This is important for

parallelisation and distribution of simulations.

Control-path parallelism can be efficiently applied to multi-entity simulation if there is either a regular geometrical ordering of the elements and if it is possible to partition the geometrical cell space Σ in mostly independent sub-partitions and regions $\{\sigma_1,\sigma_2,..\}$, ideally with a 1:1 mapping of partitions on worker processes [8]. Both FEM and CA worlds can be partitioned into independent areas. MAS simulation can be partitioned the same ways based on agent context and geospatial constraints limiting the communication (mostly) of agents to the bounds of a geospatial region, e.g., social interaction of agents is limited to the current spatial context, e.g., building spaces are independent from street areas and so on.

Each sub-partition σ_i containing a number n of entities can use primarily private memory for the state variables of the entities. Only boundary entities close to another sub-partition/region require shared memory or message-based communication implementing distributed memory. The first use-case in this paper will discuss a sliced memory architecture with shared and non-shared memory segments.

Distributed simulation requires message passing that produces a significant overhead and can effect the scaling and speed-up. A hybrid distributed-parallel approach can be used as a good trade-off, i.e., parallel simulation on the same host platform using shared memory, and distribution on multiple host platforms (clustering).

Virtual Machines

A lot of numerical software frameworks rely on native code execution. I.e., there is a set of native code libraries that performs numerical operations like matrix operations or signal transformation. Some frameworks like Tensorflow for solving ML tasks support different software front-ends (e.g., Python) and hardware back-ends (e.g., CPU and GPU), creating a significant overhead. This approach typically provides optimal performance with respect to computation time and working memory allocation. Some software frameworks utilise parallel exploitation. For a local deployment this can be sufficient, for a distributed system this can be a show stopper. Depending on the computer architecture, the byte order of numerical data formats can differ in a heterogeneous computational cluster, preventing direct exchange of numerical (and any object) data. Additionally, native code frameworks can lead to code dependency problems (like non-matching API of system libraries). Compiling from source code is not always an alternative solution (like in the R framework). The Microsoft and Apple OS do not ship with code compilers.

These limitations can be overcome by the deployment of Virtual Machines and using widely available script languages. Code execution is mostly or always done directly from device-independent text code that is compiled typically to Bytecode on-the-fly (and on demand). Python is a prominent example, but the Bytecode VM is one of the slowest VMs (1:100 compared with V8/node.js). JavaScript (JS) is another widely used scriptable programming language. A script differs from traditional complete (compiled) software that there is no start-up and main code to be provided. But in contrast to

Python, JS can be efficiently parsed (v8/Spidermonkey: more than 1M lines / second), compiled, and processed by advanced VM technologies like just-in-time (JIT) native code compilation. Googles V8 engine is a prominent example achieving nearly native code performance. And even pure Bytecode engines like Spidermonkey provide sufficient performance to perform numerical data processing including simulation and data-driven modelling with ML (see the use-case section for examples).

A main advantage of text-scriptable VMs over traditionally compiled native code processing is automatic memory management by a Garbage Collector (GC). But the GC-based memory management introduce two significant drawbacks: 1. The memory allocation and moreover the memory release of unused object occupies significant computation time that is not available for real numerical computation (overhead); 2. The automatic memory release is lazy and there is a higher average memory consumption than in user/program-controlled software; theoretically. Practically manual memory management leads to memory leaks and memory errors. But the first issue is relevant in assessing the computational power of VM-driven data processing. Finally, VMs prevent the exploitation of parallelisation. The automatic memory management and GC prevents mostly the sharing of the program state by multiple VM instances, although there is progress in parallelising VMs. The node.js platform prevents for a long time multi-threaded processing. And still is a multi-threaded VM just a set of full and independent but coupled VM instances. The start-up time and the overall memory footprint is oversized. These constrains must be considered carefully if parallel and distributed systems are composed using VMs like node.js. Fine-grained parallelism cannot be exploited efficiently.

4. Multiprocessing: Models and Architectures

4.1 Processes and Composition

This work addresses control-path parallelism primarily, although data-path parallelism using GPU co-processors and general purpose programming of GPUs can be exploited additionally. The basic data processing is executed by a sequential process performing computation within a specific data context (scope). I.e., the computational system consists of a set of sequential processes $P_{seq}=\{p_i\}_{i=1,n}$ that are processed in a given order, either control- or data-driven. In numerical and machine learning tasks, the single computational processes are (data) dependent and are typically chained, i.e., from a functional view addressing the data flow $F(x)$: $x \rightarrow p_n(p_{n-1}(...p_1(x)))) \rightarrow y$, and from a process algebraic view addressing the control flow: $P=p_1 \rightarrow p_2 \rightarrow ... \rightarrow p_n$. The sequential processes p_i are typically assigned to sub-tasks of the numerical computation, in this work called code snippets. Each process p is processed in a private data and code context, exchanging data with other processes via global variables or data queues. Each

process can be in one of five control states: {START, RUN, END, AWAIT, BACKGROUND }. A process is started either implicitely or explicitly resulting in a control state transitions START → RUN. At any time the process can wait for the satisfaction of an event condition, e.g., waiting for the completion of a communication action, resultuing in a process transition RUN → AWAIT → RUN. Process flows with synchronised process chaining can be created by defining input and output data depencies with implicit data queues or by using explicit process control statements.

The multi-process model is close to the Concurrent Communicating Processes Model (CCSP), originally introduced by C. Hoare and widely used in multi-threading and multi-processing. Each process is executed independently, typically by an independent VM instance and ideally by it's own CPU (core). Synchronisation is provided by message passing or shared memory. Concurrency is resolved by mutual exclusion locks (implicitly or explicitly).

JavaScript is strictly sequentially processed without pre-emption, but the control flow can be suspended by using promise handlers and the `await` statement inside asynchronous functions, for example:

```
async function sleep(tmo) {
  return new Promise(function (resolve) {
    setTimeout(resolve,tmo)
  })
}
async function serviceloop() {
  for(;;) {
    // service
    await sleep(1000)
  }
}
serviceloop()
```

Multiple asynchronous functions can be started, suspended by the await statement with a sequential scheduling of other waiting but ready asynchronous functions (with satisfied promises by previous call of the resolver function). But there is no parallel processing of asynchronous functions (except low-level IO tasks under the hood).

The horizontal sub-task data dependency in the functional chain prevents parallel processing of these computational processes associated with the functions. But chained data processing systems can be executed by a pipe-line approach that executes the computational processes in parallel with different chained data, i.e. there is a parallel process system $Par(\{p_i\}_{i=1,n})=p_1 \parallel p_2 \parallel .. \parallel p_n$ with a set of inter-process synchronisation using channels c connecting processes $C=\{c_i{:}p_i \rightarrow p_j\}_{i=1,m}$. This approach requires an ordered data set sequence $D=\{d(i)\}$, and is only applicable to data streams (otherwise no efficient utilisation of the different processors can be achieved). Both vertical as well as horizontal pipe-lined processing is supported by the WorkBook and WorkShell software platforms.

The JS VM does not provide any programmatical parallelisation, there is only one main control-flow, even by using "asynchronous" callback functions, Although, IO behind the wall is processed multi-threaded. But using asynchronous functions with the concept of Promises and the *await* operation enables control-flow scheduling important in message-based and synchronised multi-process systems. There can be more than one suspended function call under progress that can introduce ambiguous behaviour with multiple background suspended code snippets. To prevent hidden asynchronous function executions, the WorkBook and WorkShell provide cancellation operations.

4.2 Hierarchical Processing Architecture

This software framework combines distributed and parallel composition of large-scale systems in strong heterogeneous environments, shown in Fig. 2.

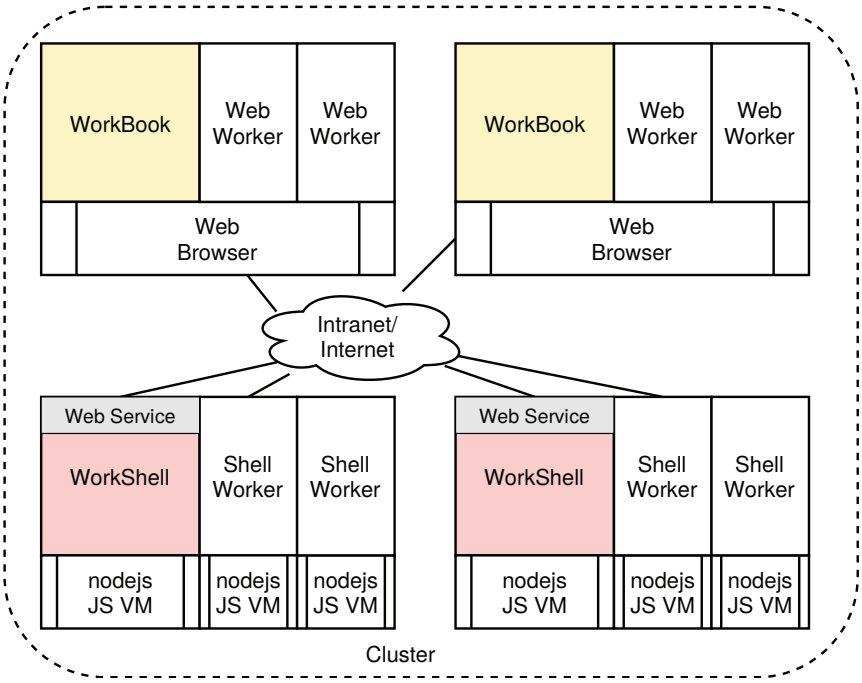


Fig. 2. The cluster approach: Heterogeneous processing architecture with Web Browsers (WorkBook) and node.js (WorkShell) connected via the Intra- and Internet, Parallel execution is provided by spawned WebWorker and ShellWorker.

4.3 Process Classes

From the technical point of view there are two different worker process classes:

1. Web Browser (WorkBook) processing JavaScript by the V8 or SpiderMonkey VMs and providing a dynamic DOM:
   - The main process (master execution and control loop);
   - Lightweight WebWorker processes (executed by OS threads controlled by the main process) with a reduced sub-set of the WorkBook (providing control-path parellelism);
   - WebGL based GPGPU kernel processes (providing data-path parallelism);
2. Node.js (WorkShell) processing JavaScript by the V8 VM:
   - The main process,
   - Separate worker processes executed by independent and isolated OS processes with a full WorkShell code base (providing control-path parallelism);
   - OpenGL based GPGPU kernel processes (providing data path parallelism).

All processes are executed by independent JS VM instances, and local and remote worker proecsse can be mxied. Inter-process Communication (IPC) is provided for unidirectional master-worker (functional data messages), unidirectional worker-master, bidirectional worker-master process RPC communication using virtual channels, and worker-worker communication via shared memory-based queues. It is assumed that a computational node $n_i \in N$ is capable to process multiple processes in parallel (multicore and/or multi-cpu architectures). Note that GPU utilisation is typically limited to one process per node and GPU!

4.4 Worker

A worker process can be created with only a few lines of program code in a Workbook or on a (remote) WorkShell directly or via the WorkShell Web API service, shown in Ex. 1 for shell workers created from a Web WorkBook remotely. There is an unified wrapper class `Worker` that covers both WorkBook and WorkShell workers as well as local and remote workers. The worker object instantiated from the `Worker` class provides full control over the worker process. The worker object can be finally used to execute code in the worker process.

```
1: // Creation of local Web or Shell workers
2: // Worker is child process of this parent process
3: var workers=[]
4: var worker = await new Worker(id?,{options});
5: await worker.ready();
6: workers.push(worker)
7:
8: // Creation of remote workers
9: // On cpu42 machine start: worksh -p 5104:protkey
10: var workers =  []
11: for(var i=0;i<NUMWORKERS;i++) {
12:   var shellworker = await new Worker('ws://cpu42:5104',i,{options});
13:   await shellworker.ready();
14:   workers.push(shellworker);
15: }
```

Ex. 1. Creation of local and remote Web and shell workers via the unified Worker class. Shell workers can be created remotely via the ShellWorker Web API.

After workers are created they can be accessed programmaticaly by executing code or by evaluating functions, shown in Ex. 2. The data state between successive function calls is preserved, i.e., functions can be state-based reusing data from previous function executions.

```
1: // asynchronous execution without reply
2: for(var i=0;i<NUMWORKERS;i++) {
3:   workers[i].run(
4:     function (i) {
5:       var result = compute(i);
6:       send(result)
7:     },
8:     i);
9: }
10: // join and collect
11: for(var i=0;i<NUMWORKERS;i++) {
12:   results.push(await workers[i].receive())
13: }
14: // synchronous execution waiting for results
15: for(var i=0;i<NUMWORKERS;i++) {
16:   var result = await
17:     workers[i].eval(
18:       function (x) { return 1/(1+Math.exp(-x)) },
19:       Math.random());
20: }
```

Ex. 2. Execution of functional code on (Web or shell) workers (following Ex. 1)

4.5 Communication, Synchronisation, and Data Sharing

Processes (either the WorkBook or WorkShell main loop, and Web or Shell worker) can communicate and synchronise by using either message-based communication M or shared memory S, shown in Fig. 3:

1. Native messages M_{native} provided by the Browser or by the node.js platform; can only be used within same process class and by parent-worker process group on the same node → Synchronisation and data transport;
2. WebSocket messages M_{ws}; can be used between all process classes and different nodes → Synchronisation and data transport;
3. Shared Memory Segments S_{sms} (SMS), implemented either with shared array-buffers in the Browser or with native externally mapped shared memory on system-level in node.js; can only be used within same process class and on the same node → Data sharing without data-driven synchronisation;
4. Shared Buffer Objects (BO) S_{smo} implemented in SMS with static typing supporting atomic core variables, structure variables, and arrays → Data sharing without data-driven synchronisation.
5. Shared matrix S_{sma} implemented by shared memory and object wrapper replications, can only be used within same process class → Data sharing without data-driven synchronisation.
6. Queued and synchronised data channels M_{ch} built on native streams, Unix or WebSocket messaging using (1/2), or shared memory (4);
7. Remote Procedure Calls over message channels.

A combination of M and S methods can be used for efficient inter-process communication in typical numerical, simulation, and Machine Learning tasks.

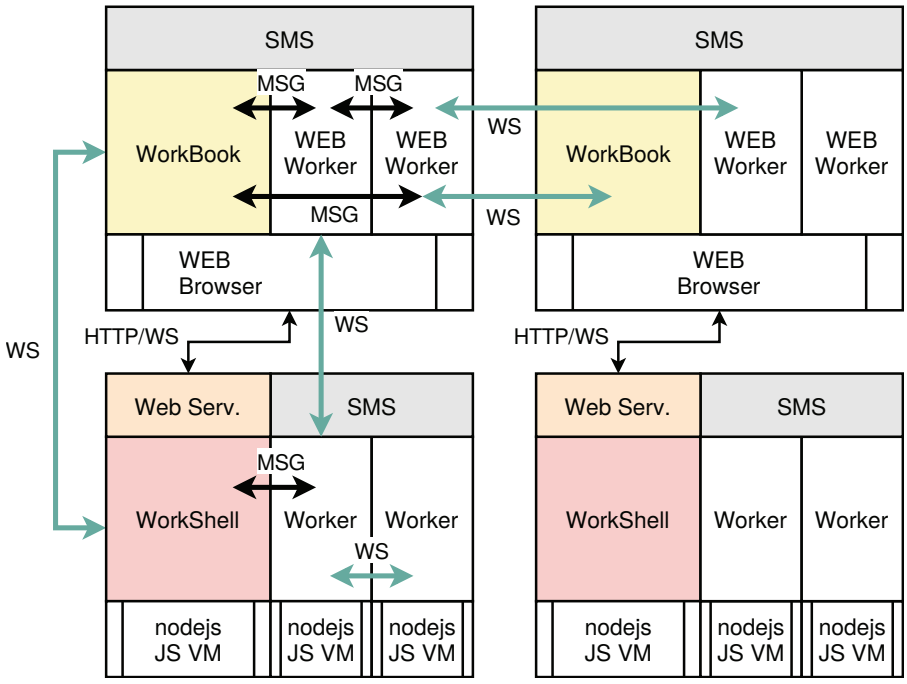


Fig. 3. Different communication methods are used to connect the cluster nodes and their worker processes. The Web Service port is used for remote worker control.

Bidirectional Master-worker communication is performed by message-based channels using *send* and *receive* operations on message channels. Additionally, a worker process can perform asynchronous (event streams) ot synchronous remote procedure calls in the master process. Worker-worker synchronisation (limited to same worker class and same host) can be established with Atomics-based operations on shared array buffers implementing the following IPC objects:

1. Mutex
2. Semaphore
3. Barrier
4. Data Queues

Local workers (instantiated from the same parent process) communicate via UNIX sockets or data pipes. Remote workers (or not instantiated from the same parent process) communicate via HTTP WebSocket (WS) channels. The protocol overhead of TCP-based WS channels is relatively low (except for the upgrade protocol), below 5-10

% depending on the payload size per message transfer, shown in Tab. 1. The constant message overhead (JSON format) is always 30 Bytes. In the case of very small payload data the overhead increases (about 20% for 80 bytes / message or 65% considering the pure payload only). The advantage of WS channels over other more efficient protocols is the capability to connect Web browser processes with remote shell worker processes (and by using a proxy server for Web worker-Web worker communication). This is a key feature for the deployment of distributed heterogeneous systems. A theoretical discussion of the protocol overhead of WebSocket connection can be found in [25]. Examples of bandwidth and message latency measurements for local worker process communication (typically via UNIX sockets or pipes) are shown in Tab. 2. The node.js platform provides low-latency and high-bandwidth communication. Current Chromium browsers pose very high bandwidth and very low latency for master-worker communication.

Host	**Matrix**	**Data (Bytes)**$\times N$	**RXTX (Bytes)**	**OF**
HPG3-WS/CFLX3-WB	2×2	(30+80) $\times$ 500000	66309928	1.21 (1.65)
HPG3-WS/CFLX3-WB	10×10	(30+1955) $\times$ 20000	42468779	1.07 (1.09)
HPG3-WS/CFLX3-WB	1000×100	(30+1929253) $\times$ 20	40784371	1.06 (1.06)

Tab. 1. Communication overhead for TCP-WebSocket channels by sending serialised matrix data (initialised with random numbers) between a shell worker and a Work-Book. The raw network RX-TX data load was measured by and collected from the LAN device. Overhead factor OF in parenthesis: Pure payload

Host	Matrix	Data (Bytes)	N	τ (ms/msg)	BW (MB/s)
CFLX3-FF52-WB	2×2	81	50000	0.37	0.21
CFLX3-FF52-WB	10×10	1943	10000	0.47	3.9
CFLX3-FF52-WB	100×100	192880	1000	8.3	22.2
CFLX6-CR90-WB	2×2	81	50000	0.01	7.0
CFLX6-CR90-WB	10×10	1945	10000	0.02	110
CFLX6-CR90-WB	100×100	192929	10000	0.42	450
CFLX3-WS	2×2	80	50000	0.018	4.2
CFLX3-WS	10×10	1950	10000	0.08	23.5
CFLX3-WS	100×100	192919	1000	1.44	127.8
HPG3-WS	2×2	81	1000000	0.005	15.4
HPG3-WS	10×10	1945	1000000	0.024	78.1
HPG3-WS	100×100	192864	10000	1.3	141.1

Tab. 2. Communication latency τ and bandwidth BW for local worker communication using channels by sending serialised matrix data (initialised with random numbers) between a worker and main process.

Channel- and stream-based WebSocket communication relies on a state-based connection using TCP between both communication endpoints. I.e., any temporary network failure or transmission disturbance result in a permanent disconnect of the communication channel. A WebSocket channel is provided by an upgrade process over a HTTP(S) connection. After a disconnect, this upgrade has to be performed again. This state-based communication is a limiting factor in the design of distributed systems and in the access of remote worker processes. Shell workers created via the WorkShell Web service are automatically terminated if there is a disconnect from the master process (e.g., a WorkBook session). All data and computation of this worker is lost permanently. To overcome this limitation there are three solutions:

1. Using state- and connection-less communication via UDP/WebRTC;
2. Workers (or the WorkerShell Web service) can be kind and tolerate temporary connection loss and wait for a reconnect;
3. A persistent snapshot by check-pointing of the worker state using secondary storage.

Solution (1) is not suitable for distributed systems. The communication channels are used for master-worker synchronisation. UDP is not reliable, there is no information if the other communication endpoint is still alive, and therefore synchronisation over such unreliable channels is not free from dead-locks and there is no starvation freedom. Solution (2) shifts the problem only temporarily, but is the most suitable method, and solution (3) results in high communication overhead. Moreover, the state of a JS execution context cannot be dumped fully (free variables cannot be resolved programmatically). But long running tasks should explicitly dump their intermediate results to en-

able an reincarnation of a crashed worker process. For example, in ML training the iteratively trained model is serialised and saved periodically. Our distributed SQL data base can be a valuable storage point. At any time the time consuming training process can be restarted from the last snapshot.

Finally, Ex. 3 shows some WorkBook/WorkShell operations to perform synchronous master-worker communication and synchronous RPC execution.

```
1: // Parent Code: Create worker with RPC handlers
2: ┌ var worker = new Worker({
3: │   rpc : {
4: │     // can be called by worker
5: │     function foo (x) { return x*x }
6: │   }
7: └ })
8:
9:
10: // Worker Code
11: ┌ var ma = Math.Matrix.Random(10,10),
12: │     mas = serialize(ma.data);
13: │ // Send data to parent process
14: │ send(mas)
15: │ var mat = deserialize(await receive())
16: │ // Execute RPC
17: └ var result = await rpc.foo(2);
18:
19:
20: // Parent Code: Get data from  worker
21: ┌ var data = await worker.receive()
22: │ var matrix = deserialize(data)
23: │ matrix.transpose()
24: └ worker.send(serialize(matrix))
```

Ex. 3. Master-Worker communication using message channels or RPC

The interprocess-communication shown above is mainly used by parent-child process groups. Totally independent worker processes can communicate via a proxy server discussed in Sec. 4.8 creating ad-hoc groups.

4.6 Security

As in any distributed system there must be some level of access protection. This addresses the access of data storage (in this framework SQL data bases with RPC service) and the access and creation of worker processes on remote machines. Typical approaches with user-related authentication and authorization is a high barrier in the implementation of heterogeneous and scalable parallel and distributed computational systems. User and authentication servers are required.

The services provided by the PSciLab software framework uses instead a capability-based access control methodologies. A capability consists of four parts:

1. A public service port assigning the capability to a specific service, but not a specific host;
2. An optional object number of the service (e.g., a specific data base or processor);
3. A rights field that specifies the allowed operations of the services (e.g., reading or writing data bases);
4. A security port that contains the encrypted rights field by using a private port.

A capability is not bound to a specific user or host computer. A capability is therefore transferable. A Directory Name Service (DNS) can be used to map names on capabilities. The private port to encrypt the rights field is handled by the service and kept secret.

4.7 Pipelines

Single code snippets can be considered as processes, too. Connecting them enables pipeline processing, shown in Fig. 4. A code snippet processes can perform computation independently from other snippets, reading input data from input channel queues, writing output data to output channel queues, and using shared state variables. In principle, the code snippet processes can be processed by different processors (web or shell worker) as long as they exchange data via the channel queues.

The queued channel network enables synchronised in-order processing of data. Beside the data exchange via channels (functional data flow), there are shared state variables that can be used by code block snippets to access data out-of-order.

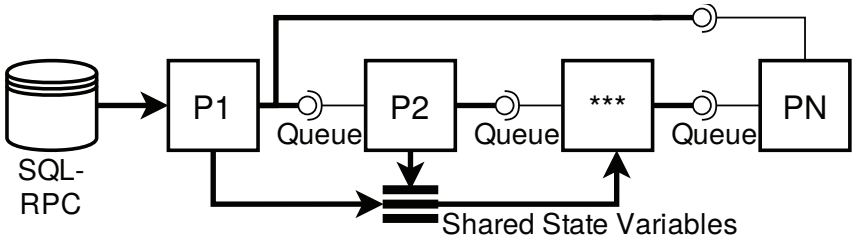


Fig. 4. Process pipelines composed of code snippet processes p_i and interconnected by channel queues. Additionally, code snippet processes can exchange data via shared state variables.

4.8 Pool Server and Working Groups

The pool server consists of a group services via HTTP(S) messages, a WebSocket proxy service with virtual circuit multiplexer (connecting groups of worker processes), and a worker management service.

The start-up time of a worker process varies between 50-1000 ms and depends on the worker class (Browser or Shell), the host platform architecture, and the OS (see Sec. 8.1). Worker processes can be created and started in advance. In this case each worker process provides an RPC service loop that can be used for control, data exchange, and code snippet execution (REPL) by a master process (or another worker). But the state of the worker process is persistent and cannot be reset, i.e., a worker proecss should not be reused for different jobs scheduled by different master processes. In contrast, worker tree clusters consist of a main process (WorkBook or WorkShell). The main process can spawn an arbitrary number of worker processes from the same class (local, i.e., Web worker or WorkShell instances) or from another class (local or remote). Each Workbook and WorkShell main instance provides a worker service that can be used to create workers.

There is a set of master main processes running in the WorkBook in a Web Browser or started by a WorkShell and executed by node.js. The WorkShell provides a Web service that enables worker creation by remote programs (assuming they have sufficient capabilities to perform this operation). The master process or an already spawned worker process can execute code on workers.

Worker processes can be grouped in pools controlled by a Pool Manager Server (PMS), shown in Fig. 5. The PMS can be accessed via Web Sockets from any host including Web Browser applications. The PMS acts as a router that can connect already started worker processes dynamically with each other and with the main process at run-time providing a virtual communication channel (VC) multiplexer. Two instance-levels of PMS registration are provided: 1. Top-level instances capable to create new workers; 2. Worker instances (already created or created on demand).

The group and pool management provides a registration service for WorkShell and WorkBook main control processes and worker processes. It performs load balancing of worker pools on new worker creation requests (process migration is not supported) or creates workers on registered WorkShells. Instead contacting the WorkShell Web API proxy service directly (although, that is still possible and useful), the pool server is responsible to select appropriate pool nodes based on static and dynamic statistical parameters (e.g., number of current worker sessions, lazy load evaluation, computational and storage metrics). The major measure of the computational power of a computing node is measured with a dhrystone benchmark (a JS version processed by the node.js or Browser VM), see Sec. 8.1 for examples. The use-case evaluation section will show that the dhrystone benchmark is suitable to predict normalised computational times. A second parameter is the number of CPUs and CPU cores of a physical node, and final-

ly the total working memory storage capacity. An 1:1 mapping of worker processes to CPUs or CPU cores per physical node should be met. The pool server tries to create and allocate worker processes in groups. I.e., a client can request a number of grouped worker processes at once with specific computational power and estimated memory storage requirements. The request can constraint the worker selection based on equally CPU measures to ensure that there is no slow worker process that suspends fast worker processes of the group to the idle state (100% load is aimed at).

WorkShell or WorkBook nodes can be added or removed at any time. Note that only a WorkShell can provide a remote Web service for worker creation (accessed remotely directly or via the pool/group services). Web WorkBooks can spawn worker processes, but cannot directly grant access to remote workers. The Web workers can only be accessed via the pool/group service.

A distributed set of PMS connected with fully meshed message channels provide suitable scaling for large-scale distributed systems. The PMS cluster handles different physical computing nodes and services independently, but negotiates worker process trading by group communication.

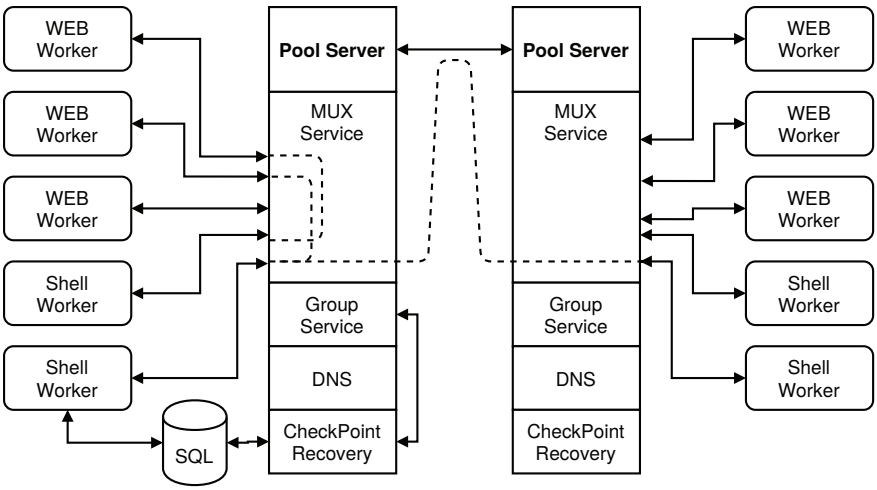


Fig. 5. The pool manager server (PMS) managing and connecting worker process groups

Load balancing in heterogeneous node clusters is important to optimise total computation times. The nodes can vary in computational power and memory capacity. The relative computation power measured in dhrystones can be used to partition computational tasks on nodes. E.g., in multi-model ML, there can be models with different number of parameters (neural nodes) and model structures. Smaller models can be processed by

slower nodes, larger models on faster nodes, still creating a significant speed-up or preserving a constant-time scaling.

Independent worker processes can create ad-hoc working groups by using the proxy and group service of the pool server. Groups can be created at run-time, and a worker process can join a group and ask for other members of the group. Each peer process is identified by a unique number and connects to the proxy service via WebSocket channels. A worker process connected to the proxy can be connected with any other connected worker process (virtual circuit channels). An example is shown below.

```
1: ┌ var gs = group.client(url,{});
2: │ var socket = await gs.connect();
3: └ gs.create('myworkinggroup1');
4: ... // wait for other members joining the group
5: ┌ var members = await gs.ask('myworkinggroup1')
6: │ // connect this peer port with other member ports
7: │ for (var i in members ) {
8: │   if (members[i]==socket.peerid) continue;
9: │   await socket.connect(members[i],true /*bidir*/)
10: └ }
11: ┌ // Communicate via socket
12: │ socket.write({cmd:'eval',f:function (x) { return x*x},
13: │               x:Math.random(),from:socket.peerid})
14: └ var result = await socket.read();
```

Ex. 4. Direct worker group programming using the proxy and group service

4.9 Worker Process Control

Worker processes are created and controlled programmatically at run-time. There is no pre-defined multi-processing network. Code snippets or functions can be executed on a worker process either by the WorkBook code snippets with a pragma code line pointing the execution to the desired (spawned) worker directly or programmatically by code processing functions sending the code to the worker processes and typically waiting for the termination or an evaluation result. A worker preserves its data state between code snippet executions. Although, web worker created by the Browser, native workers created by a shell script, and shell worker processes created remotely are handled by different modules, there is one unified top-level function `Worker` that is used to create worker processes independent of their location and host and returns a process control object.

```
1: ┌          // Local Web or Shell Worker
2: │ worker = new Worker(id?.options?);
3: │          // Remote WorkerShell Service
```

```
4:  |          new Worker('shellhost:port:capability',options?);
5:  |          // Remote Proxy Service
6:  |          new Worker('proxyhost:port:capability',options?);
7:  └
8:  ┌ // Wait until the worker is ready
9:  | await worker.ready()
10: └
11: ┌ // Execute code in worker process with and w/o waiting for results
12: | await worker.run('data={x:1,y:2}; function foo(x) { return x*x }');
13: | result = await worker.eval('return foo(100)');
14: | result = await worker.evalf(function (x) return foo(x-1) },100);
15: | data = await worker.monitor('data');  // worker proxy object
16: └ print(data.x)
17: ┌ // Terminate the worker process
18: | worker.kill();
19: └
```

Ex. 16. Typical snippet for worker creation (internally with Web workers, on remote WorkShell, and by using the pool/group server proxy service))

5. IO Data Management and Data Sharing

5.1 Distributed SQL Databases with RPC

Central to any parallel and distributed system is transparent data organisation, storage, and access, using a unified data storage and query interface. In our system architecture, a distributed SQL data base with Remote Procedure Call (RPC) access service and JSON data transfer for remote access is used to store input, intermediate, and output data in an unified way. Among flat tables, hierarchical data tables and data sets are supported. Finally, the SQL data base service supports virtual tables providing a mapping of local files on tables, e.g., a CSV table stored in a file, a *numpy* matrix, or a set of image data files. The size of the data files is not limited, and clients can access the tables row-wise sliced. The SQL data base serves as an exchange platform for parallel computational processes. The main advantage of a remote distributed data management system is the capability to access data independent of the process location and processing environment, typically limiting distribution and parallelisation in a heterogeneous like the one in this work.

The SQL database consists of the following components:

1. A native SQLite3 module for the node.js platform;
2. A JavaScript interface module that creates SQLite3 data base instances and provides a programmatical access to a data base;

3. A JavaScript JSON service API mapping JSON request to SQL statements and vice versa;
4. A JavaScript RPC service via HTTP/HTTPS or WebSockets providing JSON formatted SQL queries and some additional operations. An extended JSON format is used to support functional requests with function code. The function code consists of state-based micro operations applied to the SQL data base that are used to compose complex operations.The access can be secured by a capability-based right-key authorization mechanism;
5. A file mapper module that provides virtual SQL tables from data files, e.g., image files can be accessed as raw or converted matrix data with meta information organised as rows in SQL tables.

The data base can be stored in a single file on disk or temporarily in memory. A data base and tables in data bases can be created or deleted by any worker process regardless of their physical and logical position (if the worker provides sufficient capability rights). The memory data bases play an important role for worker-worker data exchange of intermediate computation results. Any data type can be stored in table columns including serialized JavaScript arrays and records. Finally, this SQL server provides a virtual access to data files that are mapped on SQL tables, e.g., a set of image files, a set of data matrix files in numerical python format, any many more file formats are supported. Only a YAML meta file must be provided (and the data files) that describes the mapping schema.

The SQL data bases are used for input, intermediate, and output data. In the case of ML this includes storage of serialised trained models like neuronal networks. The SQL data base also provides a storage point for snapshots created by worker process. A checkpoint with a snapshot contains all data and information to restart a crashed worker process from its last state. An example of SQL-based check pointing saving the relevant data state of a worker is shown below in Ex. 6.

```
1: // Create an ANN trainer
2: var sql = DB.sql('ws://host:port:key');
3: // Create a checkpoint tables for data state snapshots
4: var cptable = 'checkpoint-'+myGroupId+'-'+myWorkerId
5: if (!options.checkpoint) {
6:   sql.create(cptable,
7:              { time:'number', iter:'number', err:'number',
8:                model : 'blob' })
9:   var model = ML.learner(ML.ML.ANN, { options })
10: } else {
11:   var row = sql.query(cptable,'*','rowid='+options.checkpoint)
12:   var model = ML.deserialize(row[0].model);
13: }
14: while (i < maxIter || err > errThr) {
15:   var result = ML.train (model, data, { options });
```

```
16:   if (i>0 && (i % 4)==0) sql.insert(cptable,
17:     {time : time(), iter:i, err:result.error,
18:      model : ML.serialize(model) });
19:   i++; err=result.error;
20: }
21: send (ML.serializes(model))
22: // delete checkpoint table
23: sql.drop(cptable)
```

Ex. 6. Worker check-pointing of the data state enabling recovery

5.1.1 Replicated Data Bases and Tables

Since a single SQL server is a critical central instance preventing appropriate scaling of parallel and distributed applications, the RPC-SQL API provides functions to copy entire tables by a single operation. Furthermore, tables distributed on multiple SQL servers can be merged.

```
var sql1 = DB.sql('ws://hosta:porta:keya'),
    sql2 = DB,sql('ws://hostb:portb:keyb');
sql1.copy('tablename',sql2)
```

5.1.2 Capability Protection

The previously introduced capability mechanism is used to protect the SQL service, data bases, and optionally specific tables. The rights mask of the capability determines the allowed operations of the requesting worker process:

1. Reading tables and table rows;
2. Modifying tables and modifying rows;
3. Creation and deletion of tables;
4. Creation and deletion of data bases.

Alternatively for the sake of simplicity and the deployment in local networks, a simple string key can be provided associated always with the full rights mask.

5.1.3 Application Programming Interface

There is a unified data base module that can be used by any worker process to access remote data bases. The API provides one main function to create an access handle for a specific remote data base service by providing a valid URL with an access capability. The access capability specifies the allowed operations on the data base. The handle provides functions to query the data base. A handle can switch between different data bases provided by the same service port, shown in Ex. 7.

```
1: var sql = await DB.sql(url,options)
2: var databaseList = await sql.databases()
3: sql.open('mydatabase1')
4: var tableList = await sql.tables()
5: await sql.create('mytable',{$colname:$type})
6: await sql.insert('mytable',{$colname:$data})
7: var rows = await sql.select('mytable',colums:string,where:string)
8: await sql.createDatabase('mydatabase2')
9: await sql.open('mydatabase2')
10: await sql.create('mytable2',{$colname:$type})
```

Ex. 7. Typical RPC-SQL access examples

The RPC-SQL service provides access to multiple data bases via one network port. Assuming sufficient client capabilities, new data bases can be created remotely by a single operation, too. The newly created or opened data base can be served by the same network port service or by creating a new service with a different network port on the fly.

5.2 Shared Memory Segments and Buffer Objects

A Shared Memory Segment (SMS) implements structured data with Buffer Objects (BO), creating Shared Buffer Objects (SBO). The SMS provides linear memory management (linear memory block allocation and freeing) applied to the shared memory segment buffer. A buffer object is a proxy for scalar, array, or record-structured data. Each time a data element is accessed appropriate buffer handler operations transform high-level data access to low-level buffer operations including coding and decoding of data values with a linear memory model. SMS relies on *SharedArrayBuffer* objects in Web workers and shared memory segments provided by the system-level OS in shell workers, respectively. A buffer object is stored in a region of the buffer segment that can be directly accessed by the high-level programming language via object monitors (proxies or getter/setter wrappers). Buffer objects are statically typed. Supported types are JS core data types (number, static strings, boolean), C data

size-constrained types (int8, int16, int32, float, double, ..), composed record data types (records), mono-typed and static sized linear arrays, and Matrix objects based on linear typed arrays.

Finally, the SMS/BO implements shareable Interprocess-Communication (IPC) objects used for worker synchronisation (on the same physical node):

- Mutex;
- Semaphore;
- Barrier;
- Data Queue (synchronisation via two Semaphores).

The IPC objects use Atomics operations applied to shared memory arrays (inside a SMS) to satisfy inherent mutual exclusion. Shell worker process (not worker threads) use named system-level semaphores instead.

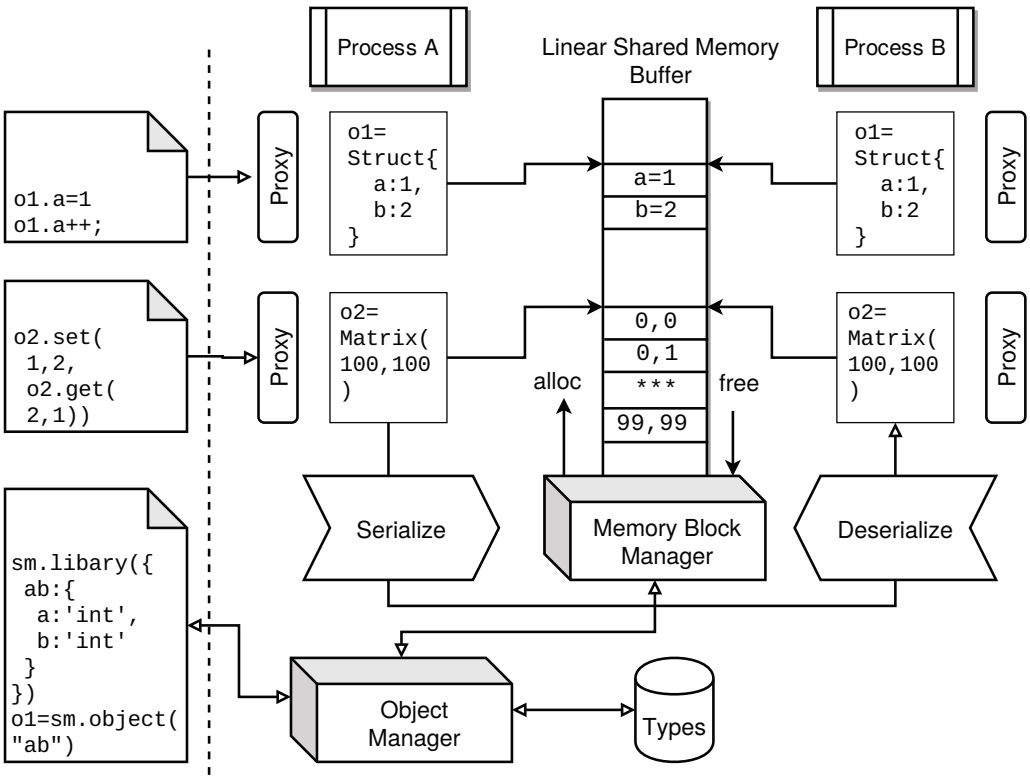


Fig. 6. Shared Buffer Objects architecture and programmatical access

There is a Memory Block manager that performs memory block allocation and release (low-level access to SMS), and a Object Manager that provides high-level access that supports object creation and destruction by using a type interface library or programmatically provided type signatures of JS objects. The overall SBO architecture is shown in Fig. 6. Any SBO can serialised in one process and deserialised in another process (assuming the SMS was already shared by the processes). The serialisation creates an object descriptor containing memory and object information (like the type signature and object class).

SMS/BO are used by multiple worker processes to implement Shared Buffer Objects (SBO) accessed concurrently. They are used to share input, intermediate, and output data between worker processes of the same class (i.e., Web worker or shell worker) and running on the same machine (host). The access of SBOs is not synchronised, although, each basic read and write access of a SBO is atomic. In contrast to message-based communication only directly possible between a worker and the main process, SBOs can be used for data communication between workers, although, there is no synchronisation. Programmatical access of SBOs are shown in Ex. 8.

Shared objects can be efficiently deployed if there is data interdependency between workers during computation. The access time of a shared object via the proxy monitor access (read/write) is only 2-10 times slower compared with a native JS object! For example, in a Web worker the read or write access of a native JS object requires about 10ns, whereas the access of a shared object (using *SharedArrayBuffer* objects) requires about 100 ns. In a shell worker using node.js the native object access requires about 2ns, whereas the shared object access (using OS-level named shared memory segments) requires about 4ns. Matrix objects that use linear typed array (shared) buffers pose similar access times. There is no significant difference in matrix cell access times (about 30 ns in a Web worker and 3 ns in a shell worker process).

```
1: // Create type interfaces
2: var typesDef = {
3:   xy : { x:'int', y:'int', z:'string' }
4: }
5: // Create a generic shared memory buffer
6: var sharedBuffer = new SharedArrayBuffer(1E5);
7: // Attach shared memory to SBS instance
8: var sm=BufferSegment(sharedBuffer,{key:'/shm1'});
9: sm.create();
10: // add type library
11: sm.library(typesDef);
12: // Create and initialize a shared object
13: objShared = sm.object('xy');
14: objShared.x=0; objShared.y=0;
15: // Create workers
16: var worker = new Worker(...)
17: // Share SMS
18: await worker.share('this.sm1',sm)
```

```
19: // Share Buffer Object
20: await worker.share('this.objShared',objShared)
21: // Shared typedarray matrix
22: var matShared = sm.object('MatrixTA-Float32',[1000,1000]);
23: matShared.set(1,1,Math.random())
24: await worker.share('this.matShared',matShared)
25: // Access shared object in worker
26: await worker.evalf(function (x) {
27:   this.objShared.x=x;
28:   this.matShared.set(1,2,
29:     this.matShared.get(2,1))},1)
30: // That's all folks!
```

Ex. 8. Creation and usage of a shared object memory with an initial set of object interfaces and one shared object

The SMS instance manages automatically memory allocation and release for shared objects using the provided shared memory segment. The SMS method `share` provides the full logic to map the shared memory to the destination worker (dependent on the worker class) and to create a copy of the SMS instance on the worker. Note that the *this* object is a persistent object in the top-level context scope of the worker and can be shared between successive worker calls.

5.3 Distributed Objects

Using object monitors and proxies it is possible to distributed objects by message passing. In contrast to the shared memory objects an additional organisational layer is required to establish group communication. Distributed objects are not considered in this work and by the PSciLab software framework, but the framework can be extended with this feature.

5.4 Object Monitors

Similar to the concept of distributed objects providing synchronised read- and write access, object monitors enable the remote access of objects (worker A) via a message-based shadow object wrapper providing a seamless proxy (worker B or main process). An object monitor can be accessed via the programming language by the same way as the remote referenced object, e.g., by using element selectors for data structures or an arrays (of numbers or structures). Although, write operations are supported in principle, there is no synchronisation or support for atomic operations. Object monitors are supported for Web and shell workers, too. The basic usage is shown in Ex. 9.

```
// In worker A
this.data = [1,2,3,4]
this.data[5]=5;
// In main process
var data = await workerA.monitor('this.data',1,false);
var sum = 0;
for(var i=0;i<data.length;i++) sum += await data[i];
```

Ex. 9. Remote object monitor usage

6. Software Framework

The software framework consists of the following core components. Development snapshots can be downloaded from the github repository [30].

Parallel WorkBook

Parallelism is achieved by using Web Worker, message-based communication, and shared memory (including low-level IPC via shared memory). Unfortunately, since 2018 CPU bugs enabling adversarial memory leakage lead to disabling shared memory in most major Web Browsers, and hence the usage of shared memory for fast worker-worker and main-worker IPC is limited to older Browsers or requires a local or remote HTTP(S) server that can pass sufficient rights via COOP/COEP headers to the WorkBook to use shared memory.

Parallel WorkShell

The WorkShell is the command line version of the WorkBook processed by the node.js program. Node.js wraps the V8 core JS VM with asynchronous and multi-threaded IO loops. Most WorkBook script code can be directly executed in the WorkShell and vice versa. Workers can be created by two methodologies:

1. Process Workers starting a new VM instance in a separate OS-level process, using message-based communication, and interprocess shared memory segments on OS-level (memory mappings with native code, not controlled by JS VM).
2. Thread Workers (available since node.js 10.5) starting a new VM instance in a thread, using message-based communication, and SharedArrayBuffers (memory shared by threads but controlled by the JS VMs).

Plugins and Libraries

There is a large number of plug-ins that can be loaded into a WorkBook (and Web worker) including an extended mathematics plug-in. There are different matrix modules inside the mathematics plugin representing multi-dimensional matrix object with generic arrays or linear typed arrays. A matrix object wraps the data container with a large set of methods that can be applied to the matrix (or vector), including common matrix

algebra. There is an extended unified ML plug-in that supports a wide range of ML models and algorithms, e.g., different kinds of decision trees, support vector machines, reinforcement learning, artificial neural networks including recurrent and convolutional networks. A GPU module can overlay matrix operations (part of the mathematics and ML library) with GPU-based replacements to speed-up computations.

The mathematics and ML libraries are already integrated in the WorkShell program and directly accessible by workers. The WorkShell can load additional libraries on demand. There is commonly a Browser and a shell node.js version of a library.

Data Management

For the data management a customised SQLite server is used that is processed by node.js, already discussed in Sec. 5.1.

Pool Management

Worker pools are managed by a pool/proxy/group server that is processed by node.js, already discussed in Sec. 4.8.

7. Data-path parallelism on GPU

Beside control-path parallelism considered in the previous sections, in this section additional data-path parallelism is considered performing general purpose computation on Graphics Processor Units (GPGPU) and the widely available WebGL/OpenGL API. Any computation that can be divided into independent data partitions like matrix computations can be processed by multiple GPU threads in parallel. But GPU transpiled algorithms are not generally faster than strict sequential code processed by the CPU. Most numerical frameworks (like Tensorflow) supporting different CPU/GPU back-end packages either use CPU or GPU computing only. The overhead of GPU transpilation and the low sequential speed of GPU processing elements can lead to a speed-up below 1. GPU transpilation and code execution produces a significant overhead that is only justified if the real computation time (and hence the computational complexity) is sufficient large compared to the overhead, including memory transfer and data code transformations. In this work and the PSciLab framework, dynamic CPU/GPU switching based on a-priori and profiling information chooses the best back-end at run-time based on complexity measures.

The dynamic switching (multiplexing) of computations between CPU and GPU based on profiling is eased by the fact that the GPU instruction code is derived from JavaScript kernel functions that are mostly equal to the Vanilla JS functions, except that the kernel function only processes a partition of the entire data. GPU access performing a computational task is typically limited to one process at once. If there are multiple worker processes that access the GPU concurrently then a serialisation of the GPU tasks occurs without any additional speed-up.

7.1 Matrix Operations

The example 10 shows matrix multiplication with complexity O(N^3) and N as the number of rows and columns of the matrix, respectively, using Vanilla JS and GPU kernel functions (compiled by the gpu.js package with WebGL/OpenGL [26]). Note that a O(N^3) $\rightarrow$ O(N^2) is only achieved if there are N parallel threads or processes.

$$\begin{aligned} \hat{a},\hat{b},\hat{c} &: \mathrm{R}^{N\times N} \\ c_{ij} &= \sum_{k=1}^{N} a_{ik} b_{kj} \end{aligned} \tag{1}$$

```
1: const gpu = new GPU.GPU({})
2: var N = 1024
3: ┌ var multiplyMatrixGPU = gpu.createKernel(function(a, b) {
4: │     var sum = 0;
5: │     var x = this.thread.x % this.constants.N,
6: │         y = Math.floor(this.thread.x / this.constants.N);
7: │     for (var i = 0; i < this.constants.N; i++) {
8: │         sum += a[y*this.constants.N+i] * b[i*this.constants.N+x];
9: │     }
10: │     return sum;
11: │ }).setConstants({N:N})
12: └   .setOutput([N*N]); // using flat array
13: ┌ function multiplyMatrixJS(a, b) {
14: │   var c = matrix(N,N);
15: │   for(var y=0;y<N;y++) {
16: │     for(var x=0;x<N;x++) {
17: │       var sum = 0;
18: │       for (var i = 0; i < N; i++) {
19: │           sum += a[y*N+i] * b[i*N+x];
20: │       }
21: │       c[y*N+x]=sum
22: │     }
23: │   }
24: │   return c
25: └ }
26: var a = matrix(N,N), b=matix(N,N)
27: var c1 = multiplyMatrixGPU(a,b) ,
28:     c2 = multiplyMatrixJS(a,b)
```

Ex. 10. Matrix multiplication $a \times b$ by Vanilla JS and GPU kernel functions (multi-threaded) assuming linear and compact Float32 TypedArray data. Both functions will return a new destination matrix c.

With respect to ML, matrix operation with complexity O(N^3) occurs in Convolutional Neural Network models. The convolution operation itself (with a kernel $\hat{K}$) has com-

plexity O(N^2), but the next (pooling) artificial neuron layer can pose complexity O(N^3) if there are N neurons in the succeeding layer. A neural network is a pipe-lined chain of functions. If neurons are arranged in layers, each layer $L_{\mathrm{i}}(n_1,..,n_{\mathrm{j}})$ depends on the computation of the previous layer $L_{\mathrm{i\text{-}1}}$. This data dependency limits parallelism on data-path level significantly. The parallelism degree can be maximal j (neurons), considering parallelism of the summation unit of neuron than maximal $j \times k$ with k as the number of neurons of the previous layer (and assuming fully connected layer interconnects).

$$
\begin{aligned}
\hat{a},\hat{c} &: \mathrm{R}^{N\times N}\\
\vec{l},\vec{m} &: \mathrm{R}^{N}\\
\hat{K} &: \mathrm{R}^{2r+1\times 2r+1}\\
\hat{c} &= \mathrm{conv}(a,K)\\
c_{i,j} &= \sum_{n=-r}^{r}\sum_{m=-r}^{r} a_{i+n,j+m}\cdot K_{n,m}\\
\vec{l} &= \mathrm{sigmoid}(\vec{\mathrm{sum}}(\hat{c}))\\
\mathrm{sum}_i(c) &= \sum_{n=1}^{N}\sum_{m=1}^{N} w_{i,n,m}\cdot c_{n,m}\\
\vec{m} &= \mathrm{sigmoid}\Big(\vec{\mathrm{sum}}\Big(\vec{l}\Big)\Big)\\
\mathrm{sum}_i(l) &= \sum_{n=1}^{N} w_{i,n}\cdot l_n\\
\mathrm{sigmoid}(x) &= \frac{1}{1+e^{-x}}
\end{aligned}
\tag{2}
$$

The program code for the light version of a CNN is shown in Ex. 11.

```
1: var gpu = new GPU.GPU({})
2: // Pipe-lined composed GPU kernel
3: var convMatrix = gpu.createKernel(function(src) {
4:   const kSize = 2*this.constants.kernelRadius+1;
5:   let sum=0;
6:   var   tx = this.thread.x,
7:         ty = this.thread.y;
8:   for(var i = -this.constants.kernelRadius;
9:           i <= this.constants.kernelRadius; i++) {
10:    var x = tx + i;
11:    if (x < 0 || x >= this.constants.width) continue;
12:    for(var j = -this.constants.kernelRadius;
13:            j <= this.constants.kernelRadius;j++) {
14:      var y = ty + j;
15:      if (y < 0 || y >= this.constants.height) continue;
```

```
16:       var   kernelOffset = (j+this.constants.kernelRadius)*kSize+
17:                             i+this.constants.kernelRadius;
18:       var weight = this.constants.kernel[kernelOffset];
19:       var pixel = src[y][x];
20:       sum += (pixel * weight);
21:     }
22:   }
23:   return sum
24: }).setPipeline(true)
25:   .setOutput([N,N])
26:   .setConstants({width:N, height:N,
27:                  kernel:kernels.boxBlur,kernelRadius:1});
28: var layer1 = gpu.createKernel(function(src) {
29:   var sum=0;
30:   for(var i=0;i<this.constants.width;i++) {
31:     for(var j=0;j<this.constants.height;j++) {
32:       sum += (1/this.constants.height*src[j][i]);
33:     }
34:   }
35:   return 1/(1+Math.exp(-sum));
36: }).setPipeline(true).setOutput([N]).setConstants({width:N, height:N });
37:  var layer2 = gpu.createKernel(function(src) {
38:   var sum=0;
39:   for(var i=0;i<this.constants.height;i++) {
40:     sum += (1/this.constants.height*src[i]);
41:   }
42:   return 1/(1+Math.exp(-sum));
43: }).setPipeline(true).setOutput([N]).setConstants({height:N });
44: var cnn = gpu.combineKernels(convMatrix,layer1,layer2,function (a) {
45:   return layer2(layer1(convMatrix(a)))
46: })
47: var input = GPU.input(new Floar32Array(..),[N,N]);
48: var output = cnn(input).toArray();
```

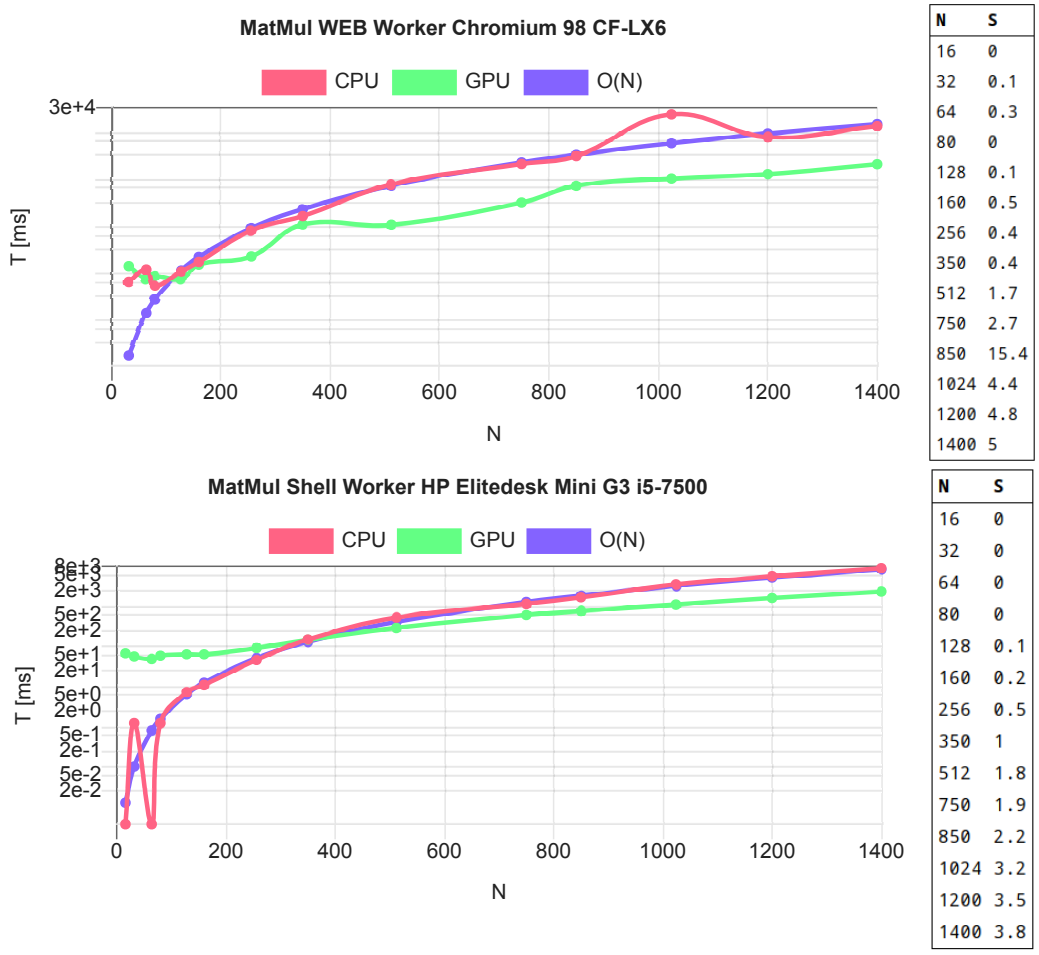
Ex. 11. The source code for the CNN light GPU test with the gpu.js library

7.2 Evaluation

The following analysis shows some selected experiments for a single matrix multiplication and a pipelined computation typical for a ML task with CNN models performing matrix convolution ($O(N^2)$) and fully connected perceptron layers containing summation and a functional application (sigmoid function) to all elements of the input matrix (vector) (up to $O(N^3)$ complexity). The figures show the computation times for CPU and GPU processing depending on the matrix size *N*. On the right side the tables show the achieved CPU/GPU speed-up.

Using generic integrated GPUs (Intel) a speed-up up to 5 could be achieved depending of the data size, CPU, JS VM, and GPU, shown in Fig. 7 and 8. The matrix data was stored in flat and compact binary Float32 arrays. The top figure analysis shows results achieved in a WorkBook worker, and the bottom figure analysis shows results achieved with WorkShell workers. Below a threshold of about *N*=300 the speed-up is below 1

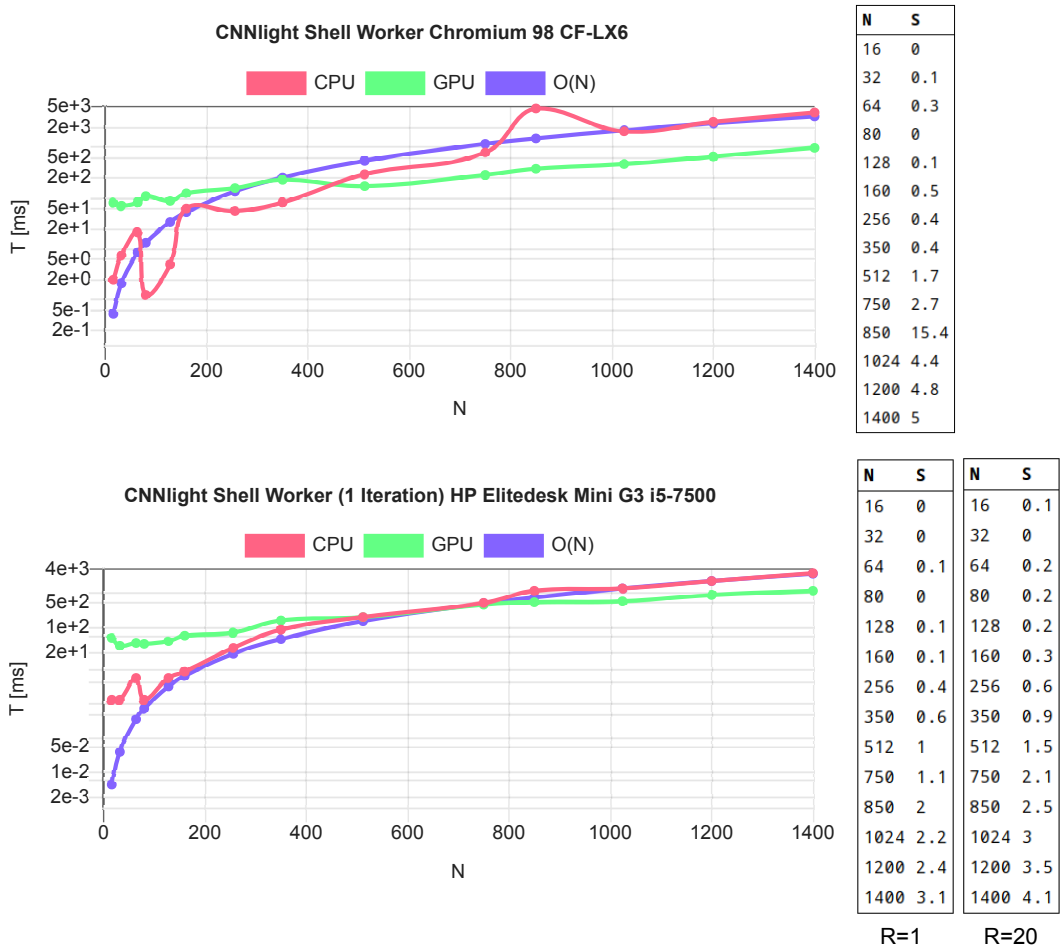
using GPU processing. The reasons are the CPU-GPU memory transfer and GPU warm-up times getting dominant for small loop iterations.



N	S
16	0
32	0.1
64	0.3
80	0
128	0.1
160	0.5
256	0.4
350	0.4
512	1.7
750	2.7
850	15.4
1024	4.4
1200	4.8
1400	5

N	S
16	0
32	0
64	0
80	0
128	0.1
160	0.2
256	0.5
350	1
512	1.8
750	1.9
850	2.2
1024	3.2
1200	3.5
1400	3.8

Fig. 7. (Top) WorkBook Worker / WebGL (Bottom) WorkShell Worker / OpenGL: Performance analysis of a $N \times N$ matrix multiplication using Vanilla JS and GPU transpiled kernel functions (Integrated Intel GPU).



N	S
16	0
32	0.1
64	0.3
80	0
128	0.1
160	0.5
256	0.4
350	0.4
512	1.7
750	2.7
850	15.4
1024	4.4
1200	4.8
1400	5

N	S
16	0
32	0
64	0.1
80	0
128	0.1
160	0.1
256	0.4
350	0.6
512	1
750	1.1
850	2
1024	2.2
1200	2.4
1400	3.1

N	S
16	0.1
32	0
64	0.2
80	0.2
128	0.2
160	0.3
256	0.6
350	0.9
512	1.5
750	2.1
850	2.5
1024	3
1200	3.5
1400	4.1

Fig. 8. (Top) WorkBook Worker / WebGL (Bottom) WorkShell Worker / OpenGL: Performance analysis of a data flow pipeline with 1 matrix kernel convolution, a fully connected percpetron layer merging the convoluted matrix with application of a sigmoid function, and a second fully connected perceptron layer (R: Number of loop iterations)

The processing time of Vanilla JS follows the expected $O(N^3)$ complexity, except in the Web Browser where is a strong degradation around N=1000 that was observed in different Browsers, too. The reason is unknown, expected L3 data cache flooding by increased matrix data size is invalidated by the decreasing execution time beyond $N >$ 1000. Maybe memory garbage collection of the VM (Google V8) has relevant impact, although there is no major data allocation during the test (all input, intermediate, and

output matrix data was allocated before computation). Up to *N*=512 the computational time using the JS VM CPU is only about two times slower than an optimized native code (C) implementation.

The pipe-lined CNN light computation shows similar results for both Web and Shell worker (host platforms with integrated Intel GPU) with a typical speed-up about 3-4. For CPU computations irrelevant, the GPU computation shows a slight speed-up if the computation is performed repeatedly ($R > 1$) in a loop without any intermediate CPU-GPU memory transfers (bottom figure, second right table), typical for ML trainings processes.

This analysis shows clearly that deployment of GPGPU for medium data-path parallelism only has a benefit for large matrix sizes and matrix loop iterations. Especially the V8 engine poses already high nearly native code performance for compact and nested loop iterations over matrix data. This observation is in accordance to the analysis performed in [16].

8. Use-cases

8.1 Test Framework

The evaluation of the PSciLab software framework with its unified processing and communication architecture with respect to speed-up and scaling by parallel processing is performed using different hardware and software platforms, summarized in Tab. 3. Both shell and Web worker classes are addressed. To compare different hardware architectures and systems with each other a dhrystone benchmark is performed. The dhrystone benchmark combines different typical high-level operations: Object allocation and release, function calls, array and string operations, and numerics. The original dhrystone was adapted to JavaScript (from an already existing dhrystone adaptation for Python). The (js)dhrystone results are important key measures for the pool server to estimate the computational power of a host and to balance worker distribution. Another important key measure is the start-up time of a new worker. Finally, a computational efficiency is computed for each computer architecture:

$$\eta = \frac{k\text{jsdhrystones} \cdot \text{NC}}{\text{TDP}} \left[\frac{1}{Ws}\right] \tag{3}$$

The Thermal Dissipation Power (TDP) is related to the CPU power consumption and another key measure to assess the computer system power for numerical computations. *NC* is the number of processor cores.

Acronym	Class	NC	jystones	T_{st}	TDP	Eff. η
HPG3	WS	4	9800k	200 ms	65 W	600
HPZ620	WS	2×6	7100k	490 ms	105/210 W	405
XEON	WS	4	11200k	300 ms	84 W	533
RP3B	WS	4	745k	5000 ms	5 W	596
CFLX3	WB	2	6700k[1] 2700k[2]	150 ms	15 W	890[1] 360[2]
CFLX6	WB	2	8200k[1] 10400k[4]	80 ms	25 W	656[1] 832[2]
G3	WW	2	140k[3]	-	5 W	56
G7	WW	4	340k[3]	-	7 W	388
XL	WW	4+4	360k[3,5]/280k[3,6]	-	5 W	576

Tab. 3. Test hardware and software platforms ([1]node.js, [2]Firefox 52, [3]Chrome 89, [4]Chrome 90, [5]: High-perf. cores, [6]: Low-perf. cores, T_{st}:Start-up time / worker, WS: WorkShell & Shell worker, WB: WorkBook & Web worker, WW: Web WorkShell & Web worker)

- HPG3-WS: HP G3 Mini, Intel i5-7500 CPU @ 3.40GHz, 8GB DRAM, 6MB L3 Cache, 4 Cores, Debian 10, WorkShell, node.js v8
- HPZ620-WS: HP Z620 Workstation, Intel Xeon CPU E5-2667 0 @2.9GHz, 2 CPU, 6 Cores/CPU, Debian 10, WorkShell, node.js v8 |
- XEON-WS: Intel Xeon Server, Intel Xeon CPU E3-1225 v3 @3.2GHz, 4 Cores, 8MB L3 cache, 16GB DRAM, Debian 10, WorkShell, node.js v8 |
- RP3B-WS: Raspberry 3B, ARMv7 Processor rev 4, BCM 2837, 4 Cores @1.2GHz, 1GB DRAM, 512kB L2 cache, Debian 10, WorkShell, node.js v10 |
- CFLX3-WB: Panasonic Notebook CF-LX3, Intel i5-4310U @3.0GHz, 8GB DRAM, 3MB L3 cache, 2 Cores, SunOS 11.3, WorkBook, FireFox 52
- CFLX6-WB: Panasonic Notebook CF-LX6, Intel i5-7300U @3.5GHz, 8GB DRAM, 3MB L3 cache, 2 Cores, Debian 10, WorkBook, Chromium 90
- G3-WB: Motorola Moto G3 smartphone, Qualcomm Snapdragon 410 8916 CPU @1,40 GHz, 4 Cores, 1GB DRAM, Android OS 5.1, WorkBook, Chromium 89
- G7-WB: Motorola Moto G7 smartphone, Qualcomm Snapdragon 632 CPU @1.8GHz, 8 Cores, 2GB DRAM, Android OS 10, WorkBook, Chromium 89
- XL-WB: Unihertz Atom XL smartphone, Helio P60 CPU @2.0GHz, 4 HP and 4 LP Cores, 2MB L3 Cache, 6GB DRAM Android OS 10, WorkBook, Chromium 89

The following two use-cases will be used to investigate the capability of parallel and distributed data processing by evaluating the speed-up S and the parallel scaling index σ.

For a static size problem based on data partitioning there is:

$$\begin{aligned} S &= \frac{T_1}{T_{PN}} \\ \sigma &= \frac{S}{PN} \end{aligned} \tag{4}$$

with *T* as the computation time with a specific parameter set and *PN* as the number of parallel worker processes. The comparison of different computers for a specific computational task requiring T_1 (in seconds) for the single-process execution can be expressed by the *R* factor:

$$R = k\text{jsdhrystones} \cdot \frac{T_1}{1000} \tag{5}$$

For a dynamic (or dynamically growing) size problem, where the number of processes *PN=DN* grows linearily with the number of independent computed data (or model) sets *DN*, there is under the assumption that all worker processes have the same workload:

$$\begin{aligned} S &= (DN) \cdot \frac{T_1}{T_{PN}} \\ \sigma &= \frac{S}{PN} \end{aligned} \tag{6}$$

The parallelisation of a static size problem with partitioning aims to reduce the total computation time, whereas a dynamic size problem aims to keep constant total computation time.

8.2 Simulation with Parallel Cellular Automata

Computation and simulation using Cellular Automata (CA) are well suited for distributed and parallel processing due to the short-range data dependencies. A rectangular and regular cell grid is assumed. The two-dimensional cell grid world can be partitioned in initially independent sub-partitions. Only partition boundaries require data exchange. Typically, a cell of a CA require only the state variables from some neighbouring cells (Moore- and von-Neumann neighbourhood with radius typically within one or two hops). Partitioning of the CA world (cell grid) enables parallel processing. The data state of a cell consists of private s and public variables S. The private state must only be shared with

neighbouring cells. And only cells near the partition boundary (with a neighbour partition) require communication by messaging or by using shared memory, shown in Fig. 9.

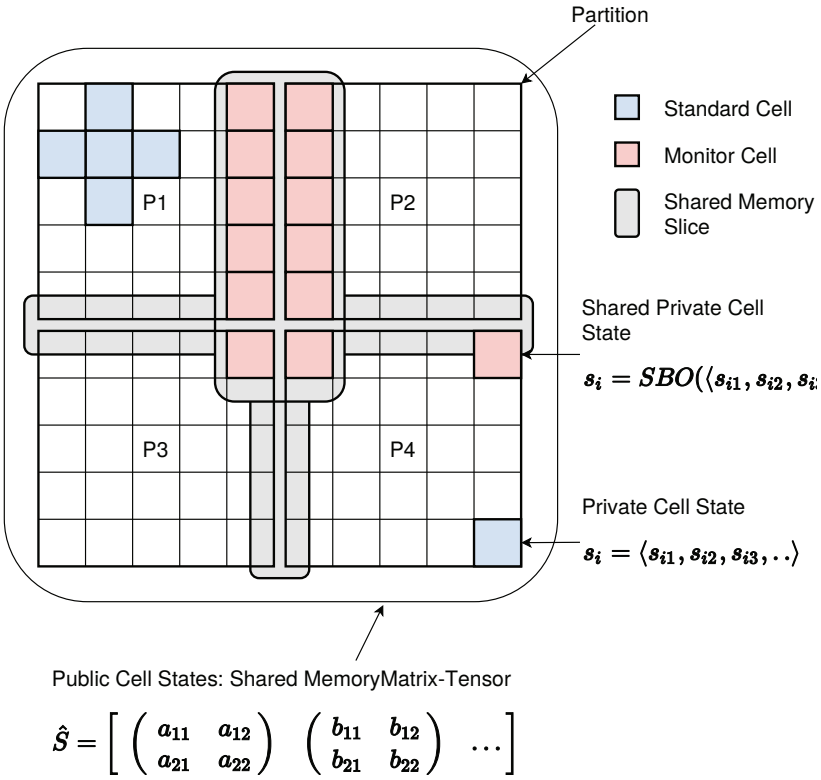


Fig. 9. Basic partitioned architecture of the CA. Each partition of the CA world is executed in a separate process. Data exchange is performed via shared memory.

The CA world is partitioned into *M* partitions. The number of parallel processed partitions depend on the total number of cells *N*. To benefit from parallel processing the communication overhead must be considered carefully, which is dependent on the communication class, too. Message-based communication is expensive, shared memory communication is more efficient, but still increases the cell access time to their own state variables.

To minimize communication and the memory complexity, private state variables are only shared with other partitions within the neighbourhood radius using slice shared memory arrays, and public (globally visible) state variables (the output of an cell) are

merged in a compact shared memory matrix, limiting the type of these state variables to atomic data types like Integer numbers.

```
var model = {
  rows:20, columns:20, partitions:[2,2],
  parameter : { P:0.3 },
  radius : 1, neighbors : 8,
  cell : {
    private : { id:0, next : null },
    shared : { place : 0 },
    public : { color : 0 },
    before : function (x,y) {
      if (this.place==0) return;
      this.next = this.ask(this.neighbors,'random','place',0)
    },
    activity : function (x,y) {
      if (this.place==0) return;
      if (this.next) this.next.place=this.id;
    },
    after : function (x,y) {
      if (this.next && this.next.place == this.id) {
        this.place=0; // we have won the competition
      }
      this.next=null;
      this.color=this.place==0?0:1
    },
    init : function (x,y) {
      this.id=1+this.model.rows*y+x;
      if (Math.random() < this.P) this.place=this.id;
      else this.place=0;
      this.next=null; this.color=this.place==0?0:1
    }
  }
}
var simu = CAP.Lib.SimuPar(model,{print:Code.print});
await simu.createWorker()
await simu.init()
await simu.run(100)
```

Ex. 12. CA random walk model used for the parallel simulator (here with 2 × 2 partitions)

The use-case is a simple particle random walk simulation. The model is shown in Ex. 12. The full source code can be found in Appendix A. A more complex example using the PSciLab software framework can be found in [27] where longitudinal and spatial infection development is studied. The results from the simple random walk simulation can be directly transferred to the more complex infection simulation. Additionally, the simulation world was partitioned into different spatial domains (e.g., home and working area), enabling parallel and distributed processing more efficiently.

The simulator processes cells in three phases that are synchronised by barriers: Before, Activity, After. This three-phase protocol is required to avoid race conditions due to concurrent modification of shared memory objects. In the first phase each cell with *place* $\neq 0$ selects one neighbour cell randomly with *place* = 0 (not occupied). In the main activity each occupied cell allocate the *next* place with their unique identifier number. In the third phase the current place occupation is only removed if the occupant id won a possible neighbour cell competition (i.e., *next.place* is equal to the current identifier number), otherwise the current occupant stays in the original cell. Thr problem size is constant, the partitioning of the CA world in smaller parallel processed sub-partitions should create a speed-up of the overall simulation time (per simulation step).

Experimental results are shown in Fig. 10 for desktop and workstation computers and in Fig. 11 fro Web Browsers. The scaling for parallel computing is under linear and the speed-up saturates, mostly due to the bottleneck of the shared memory architecture of the host platform. Even on a 2 CPu / 12 cores workstation the maximal speed-up is below 6 (expected is 12). The embedded low-resource Raspberry 3 computer scales nearly linearly up to *PN* equal the number of cores (4) and sufficient large grid worlds. The speed-up and scaling generally decreases with decreasing grid size (and number of cells per partition).

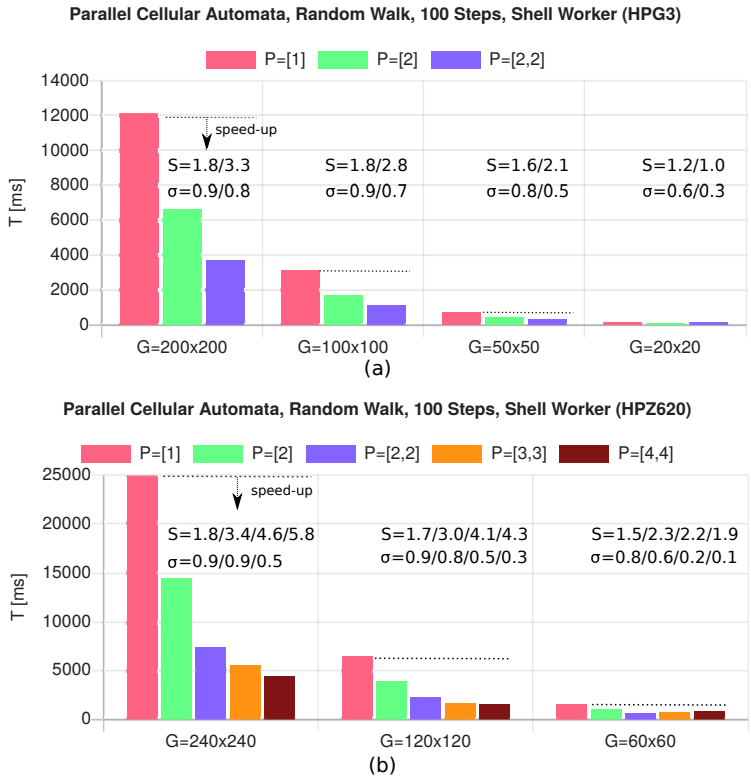


Fig. 10. Parallel simulation results using shell worker processes showing total simulation time for 100 simulation steps in dependency of the parallel partitions [rows,cols], S: speed-up, σ: scaling, (a) HP-G3 Mini (b) HP-Z620 Workstation

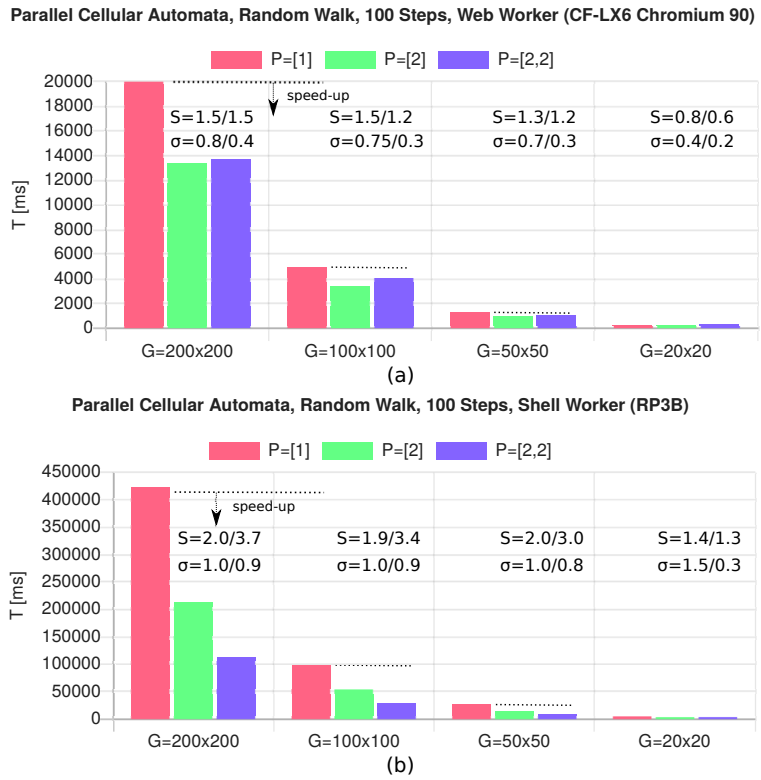


Fig. 11. (a) Parallel simulation results using Web worker processes showing total simulation time for 100 simulation steps (b) Results for shell worker processes executed on an embedded computer (Raspberry PI 3+); S: speed-up, σ: scaling,

The product of the simulation times (one process) and measured (j)dhrystone benchmark values (relative CPU scaling) R=*k jystones*•T_1(s)/1000 for a grid size of 200×200 cells is not constant over the different host architectures using the V8 JS core engine (Bytecode + just-in-time native code compilation), showing limited suitability for host selection and computational time predictions (e.g., by the pool server):

- CFLX6 (Chromium 90): R=208
- HPG3 (node.js): R=117
- HPZ620 (node.js): R=177
- RP3B (node.js) R=305

The memory architecture (shared memory bus and cache memory architecture, number of RAM access ports) of the used host platform seems to have significant impact on the performance and computational times. The next use-case does not use shared memory showing closer R values.

8.3 Multi-model and Distributed-Parallel Ensemble Machine Learning

ML model training from data is an iterative parameter optimization problem based on a set of data samples and an error (loss) function. It is difficult to parallelise the training of a single model on control-path level using workers. Therefore, in this use-case parallel multi-model training should be demonstrated and evaluated. It is assumed that there is a set of independent models. Either inherently by a distributed data problem or by creating a forest of models for best-of selection or by applying model fusion methodologies (like Random Forest Trees), shown in Fig. 12. The full source code of this use-case can be found in the Appendix A using a proxy-group server, too.

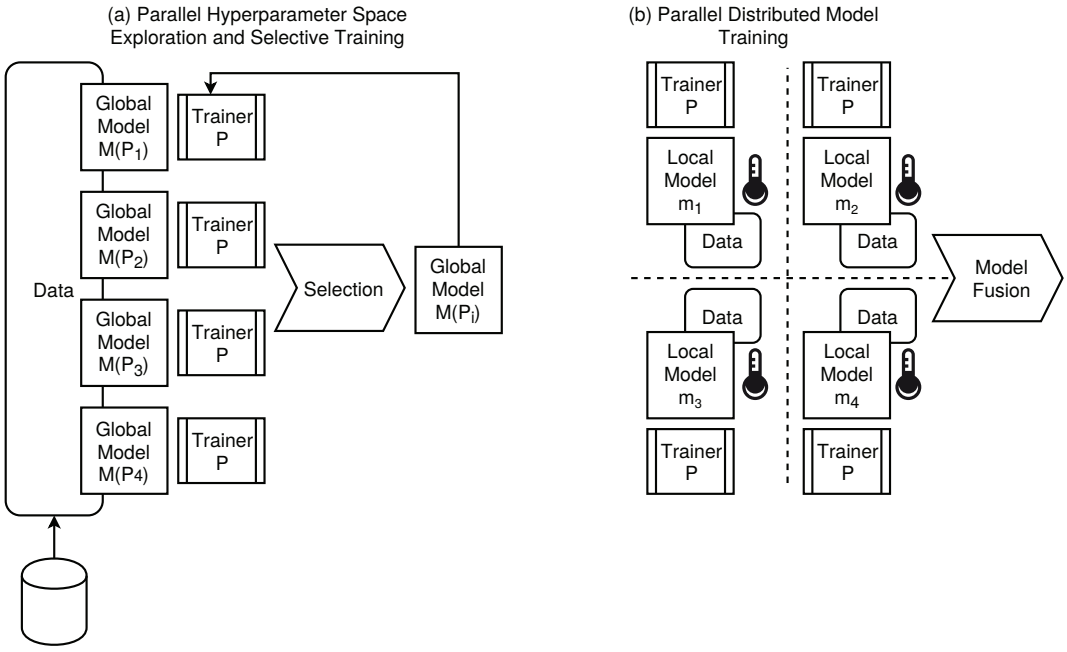


Fig. 12. Parallel multi-model training (a) Model diversity (b) Distributed models using local sensor data

Different models with different parameter settings (and/or structural complexity) can be investigated in parallel. In heterogeneous clusters, the processing nodes can vary in computational power and memory capacity. The relative computational power measured in dhrystones can be used to map computational tasks on nodes. E.g., there can be models with different number of parameters (neural nodes) and model structures. Smaller models can be processed by slower nodes, larger models on faster nodes, still preserving a constant-time scaling (dynamic size problem).

All shell workers have direct access to the extended Math and ML modules. Web worker must load the extended Mathematics and ML plug-ins first. It is assumed that the master process can create the requested worker instances on a remote WorkShell directly (or within the same Web Browser instance using Web workers).

```
1:  // Create workers
2:  for(var i=0;i<P;i++) workers[i]=
3:    new Worker('ws:shellhost..',i);  Shell worker remotely
4:    new Worker('ws:proxyhost..',i);  or Pool Proxy
5:    new Worker(i); // or Web/Shell worker locally
6:  for(var i=0;i<P;i++) await workers[i].ready();
7:  // Functions executed by workers
8:    function loadAndProcessData(OPTIONS) {
9:    var db = DB.sql(OPTIONS)
10:   for (i=1;i<ROWS;i++)
11:     data.push(await db.select(INPUTDATATABLE,'*','rowid='+i);
12:   var data = data.map(function (row) { return SQL2DATA(row) }
13:       data = data.map(function (row) { return ML.scale(row,scale) }
14:   var parts = ML.split(data,TRAINPART,TESTPART)
15:   this.dataTrain = parts[0]; this.dataTest = parts[1];
16:   send({ RESULT })
17: }
18: function createModel(OPTIONS) {
19:   this.model = ML.learner(ML.ML.ANN, { OPTIONS })
20:   send({ RESULT })
21: }
22: function trainModel(OPTIONS) {
23:   ML.train(this.model,this.dataTrain, { OPTIONS })
24:   send({ RESULT })
25: }
26: function testModel(OPTIONS) {
27:   ML.predict(this.model,this.dataTest, { OPTIONS })
28:   send({ RESULT })
29: }
30: // Execute four phases sequentailly
31: for(var i=0;i<P;i++)
32:   workers[i].eval(loadAndProcessData,OPTIONS); // async call
33: for(var i=0;i<P;i++)
34:   results[i]=await workers[i].receive(), checkResult(results[i])
35: for(var i=0;i<P;i++) workers[i].eval(createModel,OPTIONS); // async call
36: for(var i=0;i<P;i++)
37:   results[i]=await workers[i].receive(), checkResult(results[i])
38: for(var i=0;i<P;i++)
39:   workers[i].eval(trainModel,OPTIONS); // async call
```

```
40: | for(var i=0;i<P;i++)
41: |   results[i]=await workers[i].receive(), checkResult(results[i])
42: | for(var i=0;i<P;i++)
43: |   workers[i].eval(testModel,OPTIONS);  // async call
44: | for(var i=0;i<P;i++)
45: └   results[i]=await workers[i].receive(), checkResult(results[i])
```

Ex. 13. A typical ML multi-model training program with hybrid parallel-distributed worker clusters (shortened form)

The problem size in this use-case is dynamically growing. Each new worker trains a new independent model, i.e., the number of workers *PN* is equal to the parallel training of *PN* models from the same input training data. For the sake of comparability, it is assumed that the models trained in parallel differ only in their parameter state (i.e., due to randomness in the training process), but they are structural equally with same computational complexity. The training is incremental and selective. Due to stochastic initialisation of the model parameters and optionally a stochastic gradient descent (SGD) training (selecting single data instances randomly) the different models develop differently and pose different training progress given by the loss function. During parallel training, the best models are selected for further training iterations. Additionally, different model network configurations (static model parameters) can be evaluated, i.e., performing a parallel hyper parameter space exploration.

The problem used is a typical image classification problem (here with 4 different class labels). A standard Convolutional Neuronal Network (CNN) was used. Input data consist of a large set of small segment images (64×64 pixels) taken from 4K underwater images, the output is the prediction of a scene class from a set of classes (three different scen classes). The used CNN software framework was derived from the ConvNet.js framework [28]. The CNN model had the following configuration, and the trainer was an *adadelta* algorithm:

```
layers:[
  {type:'input', out_sx:64, out_sy:64, out_depth:3},
  {type:'conv', sx:5, filters:8, stride:1, pad:2, activation:'relu'},
  {type:'pool', sx:2, stride:2},
  {type:'conv', sx:5, filters:16, stride:1, pad:2, activation:'relu'},
  {type:'pool', sx:3, stride:3},
  {type:'softmax', num_classes:4}
]
trainer : {method: 'adadelta',
  l2_decay: 0.002,
  batch_size: 10}
```

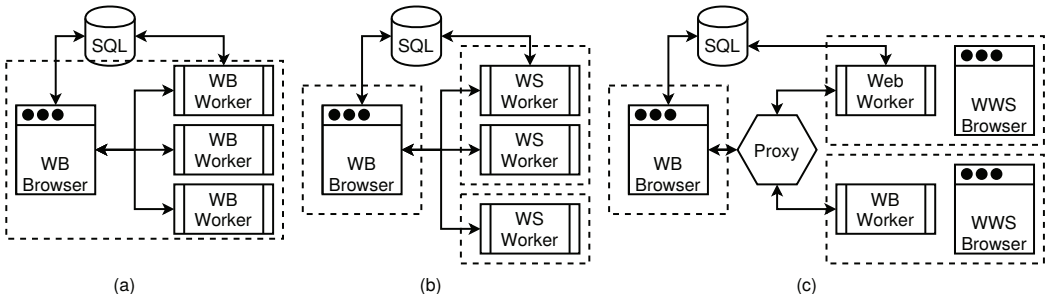


Fig. 13. The distributed-parallel worker architectures used for multi-model ML training (a) Parallel Web workers managed by a WorkBook (b) Distributed-Parallel Shell workers (c) Distributed Web WorkShell workers; WB: WorkBook, WS: WorkShell, WWS: Web WorkShell

Different worker cluster architectures were investigated, shown in Fig. 13. The entire image data set consists of about 1800 64×64×3 RGB image volumes (8 bit / element). Training and test date were split 1:1 randomly, i.e., about 900 training and 900 test images. The loading from local and remote SQL data bases, the pre-processing times, and the ML-CNN training times for one epoch (iteration) using shell workers is shown in Fig. 14.

The entire ML task is divided into four phases (the first three are considered in this experiment):

1. Loading of the input data;
2. Pre-processing of the input data (creation of training and test data, data normalisation, filtering);
3. Model training (here a CNN), commonly the major contribution to the overall computation time;
4. Test and verification.

Each shell worker process allocated about 400 MB working memory for executing all phases. The entire image data set and CNN input data required 1900×64×64×3×4 (float32) = 84 MBytes. Therefore, it is recommended to use 64bit versions of node.js to access more than 2GB memory and computers with at least 2-4 GB memory / core.

The best single-instance training performance was achieved on the embedded HPG3 platform (4 cores) with 44 ms / iteration / image data volume. The 2 CPU 6 Core HP Z620 workstation shows higher computational times for single-instance training by about 25%. The experiment with the same ANN architecture and input data was con-

ducted with the native code Tensorflow 2.0 framework using the CPU back-end (and a HP Z620 comparable host platform). The single-instance training time was in this set-up about 20 ms / iteration / image data volume set. The JS VM processing can compete, even if processed in Web browsers, shown in Fig. 15. The experiments considered parallel processing on one physical node. Up to the number of CPU cores Web browsers show appropriate scaling efficiency about 0.7. A distributed-parallel worker-tree with different physical nodes would scale linearly, too, except for the data transfer via LAN.

Maximal speed-up $S{=}PN$ is achieved in this dynamically growing size problem if the overall computation time keeps constant with increasing PN under the assumption that all processes have the same workload (computation time). The scaling (speed-up with respect to the number of parallel worker processes PN) is nearly linear up to the maximal number of CPU cores, i.e., $S \approx NC$. The total loading time of the input data (1800 images) do not increase significantly (see Figures 15 and 17 for benchmarks) up to $PN = NC$, and is independent from the SQL server location (local or remote). The remote SQL data base access only increase the loading time by a factor two (using 1Gb/s LAN and the data consumer and source nodes were connected by the same LAN switch). Note that about 10-30 CNN training iterations (epochs) were required to achieve a good prediction accuracy about 80-90%.

The processing on the Raspberry PI3B platform only allowed one training process due to memory constraints (a worker required about 300-400MB memory). The loading time of the input data from a remote SQL data base (using 1Gb/s LAN and the data consumer and source nodes were connected by the same LAN switch) required about 8.0 s, the pre-processing of the input data about 8.3 s, and the training about 670 s. This slow down relates approximately to the measured (j)dhrystone value ratio of 1:11. It is well known that node.js underperforms on ARM platforms (the V8 core engine was designed and optimized for x86/x64 platforms).

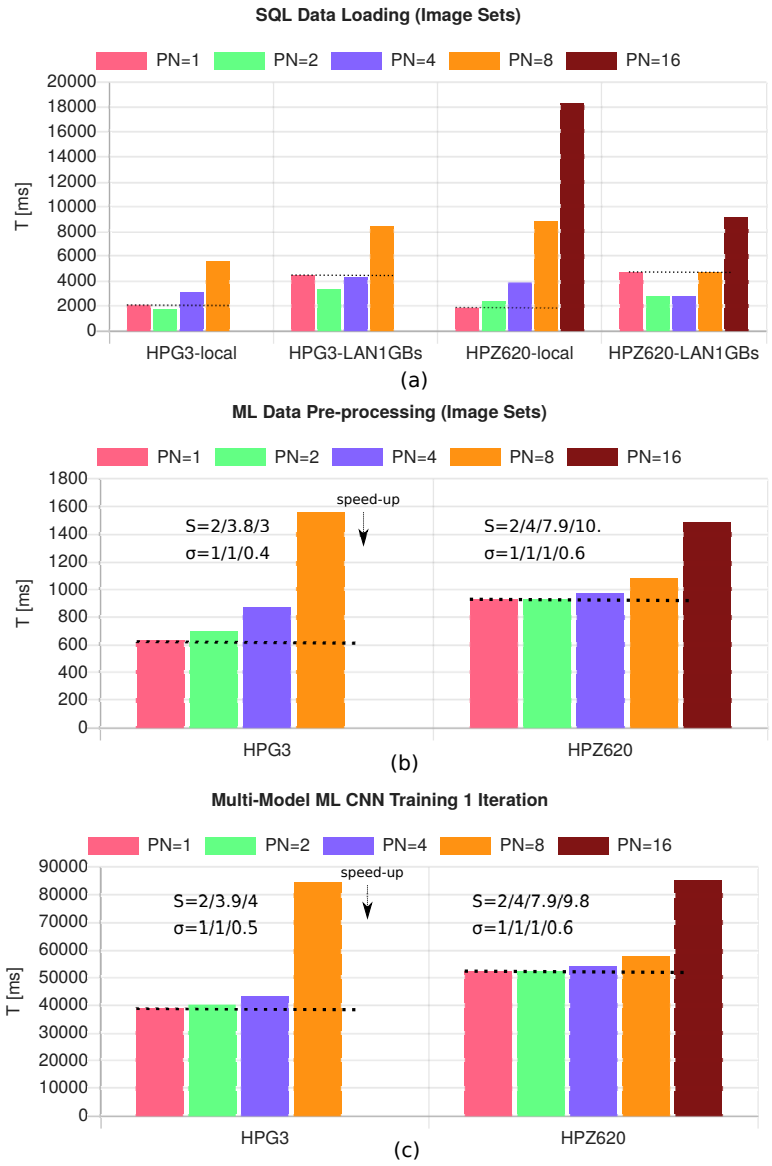


Fig. 14. Results of total computation times with shell worker processing of multi-model CNN-ML tasks with different number of workers PN. S: speed-up, σ: scaling, HPG3: NC=4, HPZ620: NC=12 (a) Input data loading times from SQL data base (b) Data pre-processing times (c) Training times

Fig. 15. Results of total computation times with Web worker processing of multi-model CNN-ML tasks with different number of workers PN. CF-LX3/LX6: NC=2 (a) Input data loading times from SQL data base (b) Data pre-processing times (c) Training times

The product of the ML training times (for a single process) and measured (j)dhrystone benchmark values (relative CPU scaling) R=*k jystones*•T_1(s)/1000 is nearly constant over the different host architectures regardless of using the V8 JS core engines (Bytecode + just-in-time native code compilation) or pure Bytecode engines (Firefox), showing the suitability for host selection and computational time predictions (e.g., by the pool server) for ML tasks:

- CFLX3 (FireFox 52): R=472
- CFLX6 (Chromium 90): R=540
- HPG3 (node.js): R=381
- HPZ620 (node.js) : R=372
- RP3B (node.js): R=498

In contrast to parallel simulation relying on shared memory processing, the scaling and performance of parallel ML tasks depend mostly on the CPU, not on the memory architecture. The V8-based Web browsers (with native code JIT support, e.g., Chrome) pose similar computation times compared with workstations and desktop computer and a speed-up of about 2-3 times compared with pure Bytecode engines (like Spidermonkey used in Firefox). Even older browser are suitable for ML computations (no significant speed-up could be observed for Firefox 73 over 52).

Two more experiments were carried out, which demonstrates the universal distributed computing capabilities of the PSciLab software framework.

The next experiment performed a hybrid distributed-parallel processing with four computer nodes (3 × 4 cores, 3 × HPG3, and 2 × 6 CPU cores, 1 × HPZ620), in total 24 CPU cores. The results are shown in Fig. 16. There is nearly a constant scaling σ=1. Note that the last first five experiments were carried out only with 3 × HP-G3 computers, and the last two with the additional HP-Z620 workstation that poses a lower dhrystone ranking (about 30% lower), which decreases the accumulated speed-up and scaling. Adjusting this ratio, the corrected scaling is still equal 1 up to 24 parallel workers.

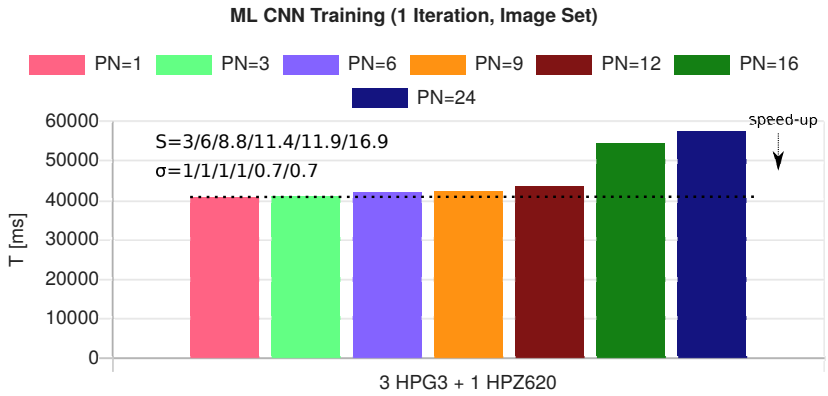


Fig. 16. Results of total computation times with a distributed-parallel cluster with shell workers processing multi-model CNN-ML traning with different number of distributed and parallel workers PN. HPG3: NC=4, HPZ620: NC=12

The last experiment exploits computational power on ubiquitous devices like smartphones. Using the Web WorkShell, one task-specific script file, the proxy server, and a WorkBook session it is possible to create a large-scale distributed computer. The proxy/group service as part of the pool server provided a WebSocket virtual circuit service, a simple group management, and an HTTP file server. All smartphones are connected via one WLAN access point (Huawei WS5200). The Web WorkShell is loaded by the smartphones via the proxy server, finally loading the script file from the proxy server, too. The script connects to the proxy server and adds their unique peer identifier to a task-specific group. Finally, the script provides a RPC service loop waiting for remote script processing (similar the RPC service provided by Web and Shell workers). The WorkBook master program processed in a Web browser distributes functional tasks to the remote smartphone workers. Results for the same multi-model CNN-ML training task using 6 smartphones are shown in Fig. 17. Note that the computational and communication power differ among the devices (see Tab. 8.1) and there can be a mix of high- and low-performance cores. There is a nearly linear scaling of the speed-up with the number of processes *PN*. Although, not expected, the speed-up scales nearly linearily with the number of cores of the smartphone. The overall performance of smartphones for numerical tasks is low compared with servers, but distributed-parallel computation can compensate this sufficiently. The *R* value for new generation smartphones (Moto G7, Atom XL) performing the ML-CNN training is lower (about 70) than on Intel x86-based systems (about 300), i.e., the ML code is executed more efficiently. The total average data loading time via the remote SQL service increases under linearily and is comparable to Web browser environments on desktop computers. The data loading time is still neglectable compared with the ML training times. Typically

10-20 epochs are required to derive a suitable model.

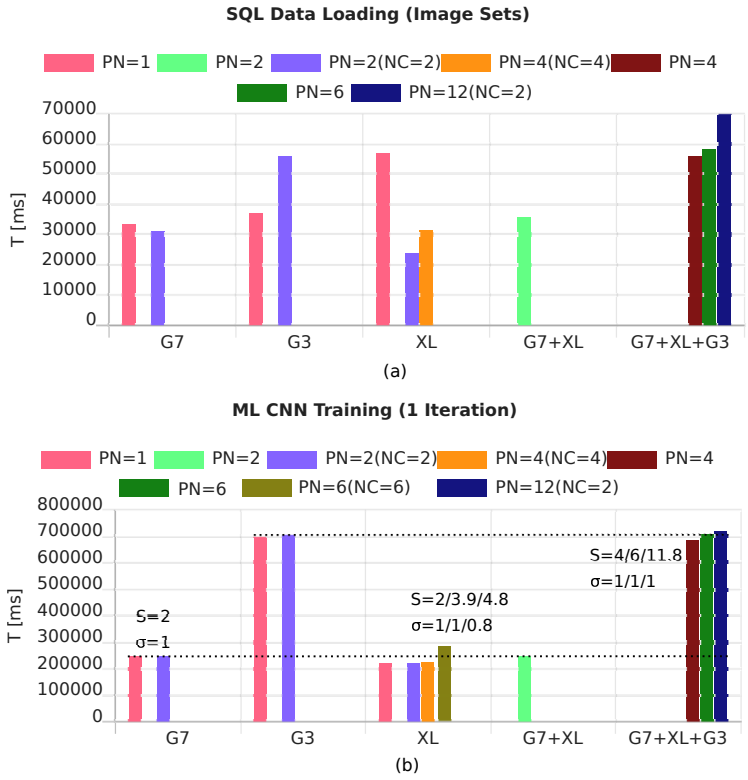


Fig. 17. Results of total computation times with a distributed-parallel cluster with Web workers processed on smartphones and performing multi-model CNN-ML training with different number of distributed and parallel workers (a) SQL data loading times (b) ML training times (1 iteration = 1 epoch)

In [29] the ML problem consisted of independent multi-instance learning and multi-instance inference with independent and partitioned input data, i.e., basically a distributed and localized multi-model ML system (finally deployed in a distributed sensor network) with global model fusion. The PSciLab framework was used to train all models in parallel. Again, a linear scaling up to the number of cores *NC* was observed (HPZ620 platform was used, too). In this use-case the problem size was static and fixed with a similar partition schema as used in the parallel simulation.

To summarise, multi-model ML training scales nearly linearly with the number of worker processes up to the number of CPU cores. Hyper-threading provide no additional improvement by using VM processing. Both Web Browser and node.js-based pro-

cessing using the V8 core VM of ML tasks perform equally. In contrast to shared-memory-based simulation, parallel ML tasks do not rely significantly on the memory architecture of the host platform. The data exchange times (input, intermediate, output) using SQLite data bases is neglectable compared with training times of medium and large-size models (typically a ration of 1:100). Even one SQLite service scales well under parallel data stream processing, multi-data-base services can improve scaling.

9. Conclusion

The introduced software framework and the processing and communication architecture is suitable to implement common computational-intensive numerical tasks like simulation and ML using Web Browsers and/or node.js as execution platforms. A hierarchical and hybrid distributed-parallel worker process composition is supported providing parallelism on control-path level with an API not requiring expert knowledge of programming of parallel and distributed systems. Worker processes can be easily created with only a few lines of code. Functional or procedural code can be executed on a worker (regardless if its a local or remote worker) by synchronous or asynchronous single operations. Data and functional code can be serialised and sent to a worker. Static and dynamic size problems are addressed and two use-cases demonstrate suitable scaling with increasing number of worker processes. Control-path parallelism can be supplemented with data-path parallelism using GPU via the WebGL/OpenGL API. Parallelism can be exploited by replacing numerical vanilla JS code with respective kernel-based GPU code using a GPGPU plug-in, but an extensive analysis showed only a medium speed-up degree for large-scale data problems.

It could be shown that a JavaScript version of the dhrystone benchmark is suitable to estimate the computational power of computing nodes for typical numerical problems like ML with message-based communication, but not accurately for problems with a shared memory architecture (like simulations). The dhrystone measures can be used by the pool server to select appropriate computing nodes. The pool server includes a virtual circuit proxy and group service. Multiple distributed pool servers can be connected.

Central part for large-scale distributed numerical systems is data storage management. Data is stored and exchanged by worker processes using a customised SQLite data base. An RPC layer was added to enable remote data base access using a JSON-SQL query format. Multiple distributed data bases can be used to distribute data base workload. Data tables can be copied between SQL data bases in advance (progressive data management). Failures of worker processes can be handled by restarting workers from a check-point state containing the relevant data. Actually it is up to the programmer to handle worker failures by doing manual check-pointing via the SQL data bases. Automatic check-pointing with snapshots optimally supported directly by the pool server have to be implemented to provide reliable distributed computing.

The entire framework and all user code is programmed with the widely established programming language JavaScript executed on Virtual Machines, most notably Googles V8 and Mozillas Spidermonkey engines. Two extended use-cases with additional references to additional scientific published work using this framework the suitability, efficiency, and mostly linear scaling of the multi-process architecture using worker processes could be shown. Worker processes can be spawned locally or remotely, either using Web or Shell workers. Code can be interchanged between both worker classes seamlessly. There is communication between master (main) and worker process established via message channels either using internal UNIX sockets or pipes (worker processes established from same master process) or WebSockets (worker processes can be established remotely). Beside message-based communication, shared memory approaches and the SBO architecture enable sharing between workers of composed objects like data records, mono-typed arrays, and matrix objects using linear buffers by providing direct high-level programming access like for any other JS object. Additionally, IPC objects like barriers are implemented via the shared memory. The high performance of message-based and shared memory communication could be shown.

The evaluation of the use-cases show high efficiency of hybrid distributed-parallel processing using widely available hardware including smartphones. Distributed processing scales nearly linearily, i.e., $S \approx PN$. The set of worker processes contains local and remote controlled processes as well as Web and shell workers, which can be mixed arbitrarily, and the processes can be executed in parallel on multi-core computers efficiently.

Smartphones using common Web browers enables in this work participative crowd computing using the Web WorkShell. But this approach requires explicit user activity and if the Web page is closed the worker processes are lost, too. The next step is the transition from participative to opportunistic (hidden) crowd computing by using service workers processing a modified head-less Web WorkShell software. Service worker are persistent and are processed beyond the lifetime of a Web page (typically up to 30 seconds) including re-incarnation. Check-pointing via the browser cache is required to survive re-incarnation and to provide scheduled long-term computations.

10. References

1. PsiLAB 1/2, *Scientific and numeric research software environment*, http://psilab.sourceforge.net, accessed 1.1.2022
2. node.js, https://github.com/nodejs/node, accessed 1.1.2022
3. Choy, R. Edelman, A., *Parallel MATLAB: Doing It Right*, PROCEEDINGS OF THE IEEE,, vol. 93, no. 2, 2008
4. Liu, X., Ounifi, H. A., Gherbi, A., Li, W., & Cheriet, M. (2020). *A hybrid GPU-FPGA based design methodology for enhancing machine learning applications performance*. Journal of Ambient Intelligence and Humanized Computing, 11(6), 2309-2323
5. Romano, J., *WebMesh: A Browser-Based Computational Framework for Serverless Applications*, Brown University, Computer Science Department, Undergraduate Research, 2019
6. Nicol, D., Fujimoto, R., *Parallel simulation today*, Annals of Operations Research 53.1 (1994): 249-285.
7. Magee, J., Dulay, N., Kramer, J., *Structuring parallel and distributed programs*, Software Engineering Journal, vol. 8, no. 2, p. 73, 1993.
8. Bagrodia, Rajive, et al. *Parsec: A parallel simulation environment for complex systems*. Computer 31.10 (1998): 77-85.
9. Kao, A., Krastins, I., Alexandrakis, M., Shevchenko, N., Eckert, S., & Pericleous, K. (2019). *A parallel cellular automata lattice Boltzmann method for convection-driven solidification*, Jom, 71(1), 48-58.
10. Rosin, P. L., *Training Cellular Automata for Image Processing*, IEEE Transactions on Image Processing, vol. 15, no. 7, 2002.
11. Giordano, A., De Rango, A., Rongo, R., D'Ambrosio, D., & Spataro, W. (2020). *Dynamic load balancing in parallel execution of cellular automata*, IEEE Transactions on Parallel and Distributed Systems, 32(2), 470-484.
12. Xia, C., Wang, H., Zhang, A., & Zhang, W. (2018), *A high-performance cellular automata model for urban simulation based on vectorization and parallel computing technology*, International Journal of Geographical Information Science, 32(2), 399-424
13. Aaby, B. G., Perumalla, K. S., Seal, S. K., *Efficient Simulation of Agent-Based Models on Multi-GPU and Multi-Core Clusters*, in SIMUTools 2010 March 15–19, Torremolinos, Malaga, Spain, 2010.

14. Xiao, J., Andelfinger, P., Eckhoff, D., Cai, W., Knoll, A., *A Survey on Agent-based Simulation using Hardware Accelerators*, ACM Computing Surveys, vol. 51, no. 6, 2019

15. Hughes, D., Correll, N., *Distributed Machine Learning in Materials that Couple Sensing, Actuation, Computation and Communication*, Cornell University Library, 2016.

16. Ma, Y., Xiang, D., Zheng, S., Tian, D., Liu, X., *Moving Deep Learning into Web Browser: How Far Can We Go?*, in WWW'19, May 13–17, 2019, San Francisco, CA, USA, 2019.

17. Teerapittayanon, S., McDanel, B., & Kung, H. T. (2017, June), *Distributed deep neural networks over the cloud, the edge and end devices*, In 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS) (pp. 328-339). IEEE.

18. Chahal, K. S., Grover, M. S., Dey, K., & Shah, R. R. (2020), *A hitchhiker's guide on distributed training of deep neural networks*, Journal of Parallel and Distributed Computing, 137, 65-76.

19. Schlegel, D., *Deep Machine Learning on GPUs*, SEMINAR TALK – DEEP MACHINE LEARNING ON GPUS, 12.01.2015,

20. NVIDIA, cuDNN Developer Guide, PG-06702-001_v8.3.2, January 2022,

21. Kotsifakou, M., Srivastava, P., Sinclair, M. D., Komuravelli, R., Adve, V., & Adve, S. (2018, February). *Hpvm: Heterogeneous parallel virtual machine*, In Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (pp. 68-80)

22. Graham, R. L., Shipman, G. M., Barrett, B. W., Castain, R. H., Bosilca, G., & Lumsdaine, A. (2006, September). *Open MPI: A high-performance, heterogeneous MPI*, In 2006 IEEE International Conference on Cluster Computing (pp. 1-9). IEEE.

23. Han, J., Haihong, E., Le, G., & Du, J. (2011, October), *Survey on NoSQL database*, In 2011 6th international conference on pervasive computing and applications (pp. 363-366). IEEE.

24. Peteiro-Barral, D. ,Guijarro-Berdinas, B., *A Survey of methods for distributed machine learning*, Prog. Artif. Intell.,2, 2013, Springer

25. Sarafov, V., Seminars FI / IITM WS 17/18, *Network Architectures and Services*, March 2018, doi: 10.2313/NET-2018-03-1_02

26. gpu.js, https://github.com/gpujs/gpu.js, accessed 1.1.2022

27. Bosse, S., *Parallel and Distributed Agent-based Simulation of large-scale socio-technical Systems with loosely coupled Virtual Machines*, Proc. of the

SIMULTECH Conference 2021, International Conference on Simulation and Modeling Methodologies, Technologies and Applications, 7-9.7.2021, Online

28. ConvNet.js, *Deep Learning in the Browser*, https://cs.stanford.edu/people/karpathy/convnetjs/. accessed 1.12.2021
29. Bosse, S., Weiss, D., Schmidt, D., *Supervised Distributed Multi-Instance and Unsupervised Single-Instance Autoencoder Machine Learning for Damage Diagnostics with High-Dimensional Data—A Hybrid Approach and Comparison Study*, Computers 2021, 10(3), 34; doi:10.3390/computers10030034
30. PSciLab software repository, https://github.com/bsLab/PSciLab, accessed on 21.2.2022

11. Appendix

11.1 Source Code Parallel Simulation

Ex. 14. Complete source code of the program code of the random walk parallel simulation (Sec. 8.2) using shared memory and worker processes. The number of worker processes is determiend by the `model.partitions` parameter array.

```
load('cap.plugin')  // load('cap.lib') for WorkShell
load('math.plugin') // only for WorkBook
// Simulation model parameters
var model = {
  rows:20,
  columns:20,
  partitions:[2,2],
  palette: ["white","red","blue","green"],
  parameter : {
    P:0.3,
  },
  size:4,
  radius : 1,
  neighbors : 8,
  name:'Random Walk',
  // Cell description
  cell : {
    private : {
      next : null,
      id   : 0,
    },
    shared : {
      place : 0,
    },
    public : {
      color : 0
    },
    // before, main, and after cell activities
    before : function (x,y) {
      if (this.place==0) return;
      this.next = this.ask(this.neighbors,'random','place',0)
    },
```

```
33:     activity : function (x,y) {
34:       if (this.place==0) return;
35:       if (this.next) {
36:         this.next.place=this.id;
37:       }
38:     },
39:     after : function (x,y) {
40:       if (this.next && this.next.place == this.id) {
41:         this.place=0; // we have won the competition
42:       }
43:       this.next=null;
44:       this.color=this.place==0?0:1
45:     },
46:     init : function (x,y) {
47:       this.id=1+this.model.rows*y+x;
48:       if (Math.random() < this.P) this.place=this.id;
49:       else this.place=0;
50:       this.next=null;
51:       this.color=this.place==0?0:1
52:     }
53:   }
54: }
55: // Create simulation world and perform the simulation steps
56: async function main() {
57:   var simu = CAP.Lib.SimuPar(model,{print:Code.print});
58:   await simu.createWorker()
59:   await simu.init()
60:   await simu.run(100)
61: }
62: main()
```

11.2 Source Code Distributed Multi-model ML

Ex. 15. Complete source code of the program code of the distributed muti-model (Sec. 8.3) using proxy and group service with remote worker processes. The number of worker processes is determiend by nodes joining the compute group and is dynamically. Required software components: workbook.htmlm worksh, webwork.html, worker.js (see next code example)

```
1: // Start proxy server:
2: // wproxy -shell webwork.html -host 192.168.0.101 -script worker.js
3: var config = {
4:   proxy : 'http://192.168.0.101:22223',
5:   workgroup : 'a5908583-31b6-46b1-bc19-1a8826d3409f',
```

```
  sql      : '192.168.0.48:9999',  // cpu48
  database : 'MariKI01',
  inputData : 'imagesSeg01',
  classes  : ["B","P","R","C"],
  // data partitioning
  trainP   : 0.5,
  testP    : 0.5,
  // training parameter
  batchSize : 10,
  l2decay   : 0.002,
}
var socket,groupControl,channelA,runCodeOnWorker
// Initialize the group, connect to proxy
async function groupInit() {
  var url = config.proxy;
  // Create group/proxy server control channel
  gs = group.client(url,{});
  groupControl = gs;
  // Connect to proxy, get socket
  socket = await gs.connect();
  // Delete, create, join computing group, ask for members (self only)
  await gs.delete(config.workgroup)
  await gs.create(config.workgroup)
  await gs.join(config.workgroup)
  await gs.ask(config.workgroup)
  // Main-Worker IPC channel
  channelA = Code.channel.create();
  // Run code on worker (function(data))
  runCodeOnWorker = function (peer,fun,data) {
    if (data==undefined) data=null;
    socket.write({
       cmd:'run',
       code:'try { var foo='+fun.toString()+'; foo(__data) } '+
            'catch (e) { print(e.toString()) }',
       from:socket.peerid,
       data:data,
    },peer)
  }
  // Process messages from worker
  socket.receiver(function (message) {
    switch (message.cmd) {
      case 'print':
        print.apply({},['['+message.id+']'].concat(message.data));
        break;
      case 'send':
        channelA.enqueue(message.data);
        break;
      case 'pong':
        print('Pong from '+message.id);
        break;
    }
  });
}
```

```
59: // Initialize the workers
60: async function initWorkers() {
61:   async function loadData(config) {
62:     var JO = JSON.parse;
63:     var t0 = time()
64:     // Create DB API
65:     this.db = DB.sqlA(config.sql);
66:     // Create or open data base, get all tables
67:     await db.createDB(config.database)
68:     await db.databases()
69:     await db.tables()
70:     // #rows of input data table
71:     var sampleN = await db.count(config.inputData)
72:     if (sampleN.length) sampleN=sampleN[0]; else return "EDBERR";
73:     this.inputData=[];
74:     for (var i = 1;i<=sampleN;i++) {
75:       // Read one input data sample (image/matrix)
76:       var data = await db.select(config.inputData,"*","rowid="+i);
77:       if (data) data=data[0];
78:       // Deserialize data
79:       data.dataspace = JO(data.dataspace);
80:       this.inputData.push(data);
81:     }
82:     // Sync. with master process, send message
83:     send(this.inputData.length)
84:   }
85:   // Get all members of the computation group
86:   var members = await gs.ask(config.workgroup),
87:       waitfor = 0;
88:   for (var i in members ) {
89:     if (members[i]==socket.peerid) continue;
90:     // Run code in group workers
91:     runCodeOnWorker(members[i],loadData,{
92:       id:members[i],
93:       index:i,
94:       sql : config.sql,
95:       database : config.database,
96:       inputData : config.inputData,
97:     })
98:     waitfor++;
99:   }
100:  for (var i=0;i<waitfor;i++) {
101:    // Wait for workers finished their work
102:    await channelA.receive()
103:  }
104: }
105: async function connectGroup() {
106:   var members = await groupControl.ask(config.workgroup)
107:   // Connect all workers to this master process (on proxy)
108:   for (var i in members ) {
109:     if (members[i]==socket.peerid) continue;
110:     print(members[i])
111:     await socket.connect(members[i],true /*bidir*/)
```

```
112:   }
113: }
114: // Pre-process the image data and create tarining and test data tables
115: async function preProcess1(config) { try {
116:   var int = function (x) { return x|0 }
117:   var t0=time()
118:   // 0. Filter out '?' labelled samples
119:   var data = this.inputData.filter(function (row) { return row.label!='?' });
120:   // 1. XY transform
121:   this.dataXY=data.map(function (row) {
122:     return { x: new Float32Array(row.data), y:config.classes.indexOf(row.label) }
123:   })
124:   // 2. Scale to [-.5,.5]
125:   this.dataXY.forEach(function (row) {
126:     row.x=ML.scale(row.x,Math.scale1(0,255,-.5,.5))
127:   });
128:   // 3. Shuffle sample instances randomly
129:   this.dataXY=this.dataXY.shuffle();
130:   // 4. Split the sample instances (training/test data)
131:   var parts = ML.split(this.dataXY,int(config.trainP*this.dataXY.length))
132:   this.dataTrainXY=parts[0];
133:   this.dataTestXY=parts[1];
134:   this.dataTrainX=this.dataTrainXY.pluck('x');
135:   this.dataTrainY=this.dataTrainXY.pluck('y');
136:   this.dataTestX=this.dataTestXY.pluck('x');
137:   this.dataTestY=this.dataTestXY.pluck('y');
138:   this.dataX=this.dataXY.pluck('x');
139:   this.dataY=this.dataXY.pluck('y');
140:   // Sync. with master process
141:   send(this.dataXY.length)
142:   } catch (e) { print(e.toString()); send({error:e.toString()}) }
143: }
144: async function preProcess() {
145:   var members = await groupControl.ask(config.workgroup),
146:       waitfor = 0;
147:   for (var i in members ) {
148:     if (members[i]==socket.peerid) continue;
149:     runCodeOnWorker(members[i],preProcess1,{
150:       id:members[i],
151:       index:i,
152:       classes : config.classes,
153:       trainP : config.trainP,
154:       testP : config.testP,
155:     })
156:     waitfor++;
157:   }
158:   for (var i=0;i<waitfor;i++) {
159:      await channelA.receive())
160:   }
161: }
162: // Create CNN models on the worker (and the trainer)
163: async function model(config) {
164:   this.model = ML.learner({
```

```
    algorithm:ML.ML.CNN,
    layers:[
      {type:'input', out_sx:64, out_sy:64, out_depth:3},
      {type:'conv', sx:5, filters:8, stride:1, pad:2, activation:'relu'},
      {type:'pool', sx:2, stride:2},
      {type:'conv', sx:5, filters:16, stride:1, pad:2, activation:'relu'},
      {type:'pool', sx:3, stride:3},
      {type:'softmax', num_classes:config.classes.length}
    ],
    trainer : {method: 'adadelta',
               l2_decay: config.l2Deacy,
               batch_size: config.batchSize}
  });
  send(1)
}
async function createModels() {
  var members = await groupControl.ask(config.workgroup),
      waitfor = 0;
  for (var i in members ) {
    if (members[i]==socket.peerid) continue;
    runCodeOnWorker(members[i],model,{
      id:members[i],
      index:i,
      classes : config.classes,
      l2Deacy : config.l2Deacy,
      batchSize : config.batchSize,
    })
    waitfor++;
  }
  for (var i=0;i<waitfor;i++) {
     await channelA.receive())
  }
}
// Train one model in the worker
async function trainOne(config) {
  var result = ML.train(this.model,{
    x:this.dataTrainX,
    y:this.dataTrainY,
    width  : 64,
    height : 64,
    depth  : 3,
    iterations:config.iterations,
    verbose : 0,
    async : false,
    callback : function (result) {
      print(result.iteration,result.loss,result.time)
    }
  })
  send(result)
}
// Parallel Multi-model training
async function trainAll () {
  var members = await groupControl.ask(config.workgroup),
```

```
218:       waitfor = 0;
219:    for (var i in members ) {
220:      if (members[i]==socket.peerid) continue;
221:      runCodeOnWorker(members[i],trainOne,{
222:        id:members[i],
223:        index:i,
224:        iterations:1,
225:      })
226:      waitfor++;
227:    }
228:    for (var i=0;i<waitfor;i++) {
229:       await channelA.receive())
230:    }
231: }
232: // Main master process function
233: async function main() {
234:    await groupInit();
235:    await connectGroup();
236:    await inirWorkers();
237:    await preProcess();
238:    await createModels();
239: }
240: main()
```

Ex. 16. Worker code provided by the wproxy server and processed by the webwork.html App in a node Web browser. This worker waits for code execution requests from the master.

```
 1: var config = {
 2:    PN:4,
 3:    proxy : 'http://192.168.0.101:22223',
 4:    workgroup : 'a5908583-31b6-46b1-bc19-1a8826d3409f',
 5:    nodeid : Utils.UUIDv4(),
 6:    verbose : 1,
 7: }
 8: // Main worker RPC loop waiting for code evalulation requests
 9: async function workerLoop (config) {
10:    print('WebWorker started.',config.workgroup);
11:    // Load plug-ins
12:    load('math.plugin.js')
13:    load('ml.plugin.js')
14:    await sleep(1000);
15:    // print(typeof group)
16:    var url = config.proxy;
17:    // Create group/proxy control channel
18:    var gs = group.client(url,{id:config.workerid});
```

```
19:   print('ping',url,inspect(await gs.ping()))
20:   // Connect to proxy server, get a communication socket
21:   var socket = await gs.connect();
22:   // Join computational group, get all members
23:   await gs.join(config.workgroup)
24:   // Returns UUID array
25:   await gs.ask(config.workgroup)
26:   var Env = {
27:     print:function () {
28:       print.apply({},Array.prototype.slice.call(arguments));
29:       socket.write({cmd:'print',
30:                     id:config.workerid,
31:                     data:Array.prototype.slice.call(arguments)});
32:     },
33:     send:function (data) {
34:       socket.write({cmd:'send',data:data,from:socket.peerid});
35:     },
36:     sleep:sleep,
37:     schedule:schedule,
38:     time:Date.now,
39:     DB:DB,
40:     JSON:JSON,
41:     ML:ML,
42:     Math:Math,
43:     Utils:Utils,
44:   }
45:   for(;;) {
46:     // Get requests from master
47:     var rpc = await socket.read();
48:     switch (rpc.cmd) {
49:       case 'run':
50:         // Run code here!
51:         try { compile(rpc.code,Env,rpc.data) }
52:         catch (e) { print(e.toString()) };
53:         break;
54:       case 'stop':
55:         Code.interrupt=true;
56:       case 'ping':
57:         socket.write({cmd:'pong',id:socket.peerid});
58:         break;
59:     }
60:   }
61: }
62: // Start the worker code...
63: async function main() {
64:   var workers=[];
65:   for(var i=0;i<config.PN;i++) {
66:     var worker = Code.worker.create(i);
67:     await Code.worker.ready(worker);
68:     print('Worker #'+i+' started.');
69:     workers.push(worker);
70:   }
71:   for(var i=0;i<config.PN;i++) {
```

```
72:     Code.worker.evalf(workers[i],workerLoop,{
73:       id:i,
74:       proxy:config.proxy,
75:       workgroup:config.workgroup,
76:       nodeid:config.nodeid,
77:       workerid:config.nodeid+'-'+i,
78:       verbose:config.verbose,
79:     })
80:   }
81:   print('Initialized.')
82: }
83: main()
```