Job-Scheduling of Distributed Simulation-Based Optimization with Support for Multi-Level Parallelism

Peter Nordin[1] Robert Braun[1] Petter Krus[1]

[1]Department of Management and Engineering, Linköping University, Sweden,
{peter.nordin,robert.braun,petter.krus}@liu.se

Abstract

In many organizations the utilization of available computer power is very low. If it could be harnessed for parallel simulation and optimization, valuable time could be saved. A framework monitoring available computer resources and running distributed simulations is proposed. Users build their models locally, and then let a job scheduler determine how the simulation work should be divided among remote computers providing simulation services. Typical applications include sensitivity analysis, co-simulation and design optimization. The latter is used to demonstrate the framework. Optimizations can be parallelized either across the algorithm or across the model. An algorithm for finding the optimal distribution of the different levels of parallelism is proposed. An initial implementation of the framework, using the Hopsan simulation tool, is presented. Three parallel optimization algorithms have been used to verify the method and a thorough examination of their parallel speed-up is included.
Keywords: Job-scheduling, parallelism, distributed simulation, optimization

1 Introduction

Design optimization is a powerful development tool, which can greatly increase the benefits of simulation. However, the usability is often limited by long execution times. A common solution is to utilize parallel execution, for example by using parallel optimization algorithms. Each simulation model in itself may, however, also be parallelized. Furthermore, multiple optimization jobs can also be executed simultaneously. To benefit the most from these methods, an intelligent scheduler for multi-level parallelism is required.

In many organizations the overall computer performance is poorly utilized a large part of the time. Computers are often used for tasks with low requirements, such as word processing, and spend a lot of time in idle mode.

This work presents a framework, illustrated in Figure 1, utilizing available network computers for multi-level parallelism during optimization. Job-level, algorithm-level and model-level parallelism is supported. The most beneficial combination of these will depend on properties of the model as well as the current state, performance and availability of simulation-service providers on the network. The algorithm used for optimization will also have a large impact on performance. For this reason, a thorough investigation of three parallel optimization algorithms is included. Two of these are custom made for the sake of the experiments, while one is state-of-the-art and naturally parallel.

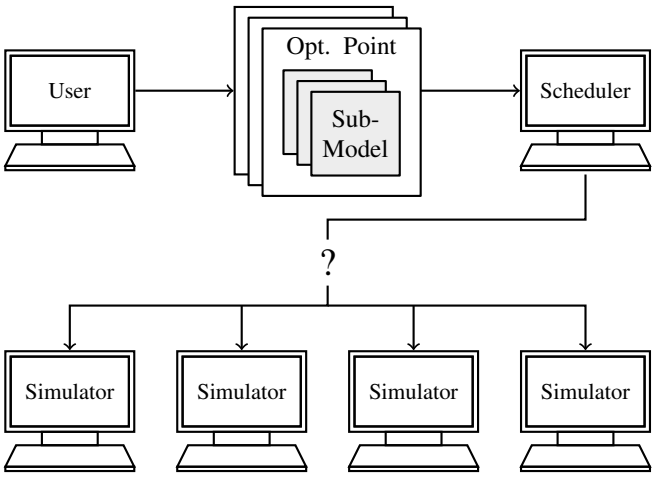

Figure 1. The core of the framework is the job scheduler. It must decide how jobs should be distributed based on available simulation resources and their performance.

The intention is to create a small-scale software with few dependencies that can easily be deployed on an office network. One goal is to support simulation models from different tools through the use of Functional Mock-up Interface (FMI) for co-simulation (Blochwitz et al., 2009).

In (Sadashiv and Kumar, 2011), a thorough comparison between the cluster and grid computing concepts is given together with examples on tools and simulation environments for such systems. Computing clusters are often built up of a homogeneous collection of computer nodes located at the same place accessed through a front-end. From the outside, usually only the front-end is visible. A

job-queuing system is responsible for managing jobs and distributing the workload among the nodes.

In a grid network, computers can be located at different places, but they all dedicate some computational power to a common governing system, a middle-ware, responsible for assigning jobs to available resources. One example of such a system is the volunteer-computing framework BOINC (Anderson, 2003). Users around the world can download a client and choose from many projects which to volunteer resources to. Grids are typically heterogeneous by nature. The participating computers may have vastly different performance and availability.

Using dedicated computing clusters would of course be beneficial, as the computational performance and availability would be reliable. Work stations on an office network should, however, be considered as a small-scale grid with a less reliable source of computational power. Computers may come and go sporadically and their perceived performance may at any time be reduced due to workload caused by their respective local user.

For parallel optimization, a third party scheduler could be problematic. Most algorithms require all parallel tasks to run synchronized. They should also have about the same execution time, since the overall speed will be limited by the slowest task.

1.1 Related Work

In (Fourer et al., 2010), a framework for distributed optimization as software services is presented. The authors identify the need for and develops standardized protocols to enable different types of optimization services (different solvers, high-level modelling languages and architectures) to register in a common registry service. The framework is built on the XML format and Web Service standards. The work focus on problems expressed in common high-level modeling languages for optimization. In contrast our work focuses on simulation based optimization of non-linear black-box models particular to specific simulation tools. Such models are difficult to generalize into high-level optimization languages. Instead of letting servers provide optimization algorithms, they provide access to one or more simulation tools.

In (Gehlsen and Page, 2001), a Java language specific framework for distributed discrete-event simulation optimization is presented. A Genetic Algorithm (GA), which is naturally parallel, is used. An optimization manager constructs and evaluates designs. An experiment manager constructs simulation tasks that are sent to a distribution manager using Java Remote Method Invocation (RMI) to simulate the tasks on available remote machines running an instance of the simulation tool.

In (Yücesan et al., 2001), a Java and web-based simulation optimization interface is presented. It allows the user to choose combinations of optimization algorithms and simulation models from databases. The key contribution is the Optimal Computation Budget Allocation (OCBA) algorithm, which allocates simulation jobs to available simulators, called the Computation Budget. New promising designs that based on statistics are likely to improve the overall simulation quality are prioritized. The total number of simulations is reduced as evaluation of non-promising designs is avoided.

The referenced works all provide a registry of simulation servers that clients can fetch available simulation resources from. A central manager then decides how to distribute the work. Clients communicate either directly with the servers in a peer-to-peer manner or by relaying all communication through the manager. The framework presented in this paper uses a similar approach.

A different approach is presented in (Eldred et al., 2000) where the challenges and possibilities of utilizing multiple levels of parallelism on massively parallel super computers have been investigated. They define four levels of parallelism, *algorithmic coarse-grained*, *algorithmic fine-grained*, *function eval. coarse-grained* and *function eval. fine-grained*. The presented implementation made it possible to recursively divide the workload at each parallelism level. That is, at the highest level only coarse-grained parallelization of the optimization algorithm is regarded but at the next lower-level each one of the previously parallelized parts do their own parallelization and scheduling into the next lower level.

For this paper, the first and last of those parallelism levels are the most relevant. Coarse-grained algorithm parallelism represents evaluation of multiple parameter sets with the same model concurrently, or running multiple optimization jobs at the same time (algorithm-level or job-level). Fine-grained function evaluation represents parallelization within a simulation model (model-level).

1.2 Delimitations

Many simulation tools use variable step-size control. Their execution time will therefore depend on the model parametrization. As a consequence, the points in the parameter space may require different amounts of time to evaluate. This variation is, however, difficult to predict. Fine-grained model parallelism through the use of the transmission line element method in Hopsan can also reduce such effects. For this reason, it has not been considered further.

Coarse-grained model parallelism, i.e. spreading the model evaluation over multiple servers, has been left for future work. This becomes beneficial when a model contains individual sub-models with a long evaluation time, making communication overhead negligible. It would also be required if models contain sub-models from different disciplines and co-simulation using different tools is needed.

Experiments have only been performed on homogeneous computer networks. In principle, the proposed algo-

DOI
10.3384/ecp15119187

rithm should work similarly on heterogeneous networks. Verifying this with experiments remains a future project.

The Hopsan tool supports importing co-simulation FMI sub-models, so called Functional Mock-up Units (FMUs). While the goal is to support such models directly in the framework, a Hopsan model must currently be used as a master.

2 Framework Architecture

The idea is to run one server providing access to one or more simulation tools on each available computer in the network. Each server provides a number of simulation slots that represent the processor cores made available.

When a simulation server is started it contacts an address server to register which simulation tools it provides. The address server will request status periodically to keep track on availability, speed and the number of available simulation slots.

When a client requires access to remote simulation, it first requests a list of available servers matching some requirements on speed and available number of slots. Then it connects to each desired server individually to request and reserve the needed slots. If a request is granted, the server will launch a separate simulation worker process. All further communication will then be directly between the client and the worker.

The reason for using an external worker process is to keep the server alive even if the simulation would lead to a crash or lock-up. The worker will report back when it is finished so that the server can reopen the simulation slots. If the worker stops responding due to some failure, the server can terminate it by force to free up resources.

The intention is to use tool specific implementations of optimization algorithms. The optimization is thus assumed to be handled by the client. The job scheduler is not part of the optimization algorithms. The tool running the optimization can, however, use it to run the model evaluations.

2.1 Decentralized and Centralized Modes

The framework can work in two modes. If no centralized job distributor is used, each client is responsible for requesting fresh server status before trying to reserve a slot. The information from the address server may be outdated. Clients send a *request and reserve* command to temporarily reserve the needed slots. An actual request can then be sent once the client has determined the work distribution. This prevents other clients from stealing a seemingly open slot. In this decentralized case, illustrated by Figure 2, clients obtain resources on a first-come-first-serve basis. A client will take as many servers and slots as it needs. Job-level parallelism is only possible as long as there are free slots. The decision whether to use algorithm parallelism, model parallelism or both is take by each client.

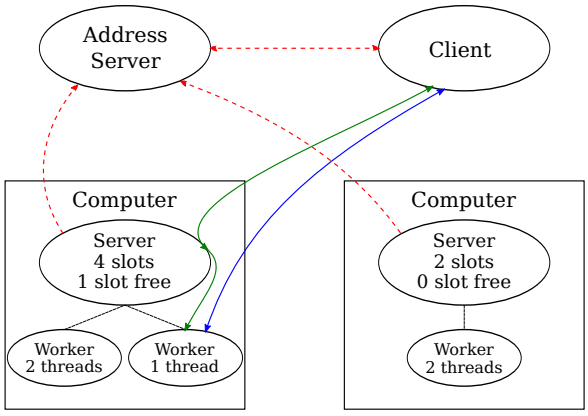

Figure 2. In the decentralized network, clients communicate directly with the servers and the simulation worker processes. A client might look up servers from a predefined list or request them from an address server.

If a centralized job distributor is used, all slot requests go through this master, which keeps track of available resources. In this case, shown in Figure 3, the clients should let the job distributor choose which servers to use. When all servers are busy, job-level parallelism could still be allowed by redistributing ongoing work when new jobs arrive. A system for prioritizing jobs is, however, needed in this case.

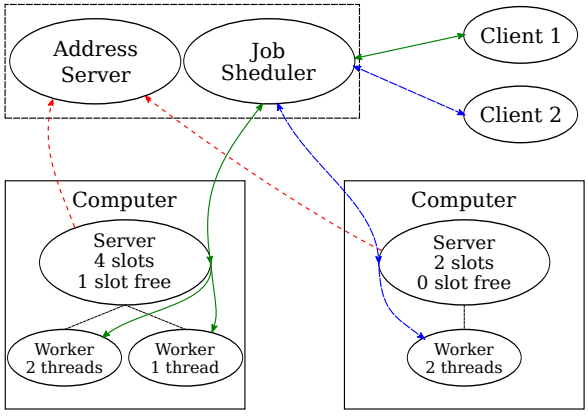

Figure 3. In the centralized network, the job scheduler is responsible for allocating jobs to simulation slots. Clients have no information about available servers and all communication is relayed through the job scheduler.

2.2 Cluster Nodes and Subnets

Simulation servers connected to a Local Area Network (LAN) can report their IP address and port directly to the address server. If a cluster or a subnet behind a firewall is present on the network, the available cluster nodes or subnet computers must either be exposed through port forwarding on the front-end computer or router, or by a relaying address server as illustrated in Figure 4. In this framework it is not possible for the front-end to expose simulation slots from the underlying servers directly. That would

confuse the scheduler and make it interpret the simulation slots as a possibility for model parallelism. Such parallelism is currently only supported within one machine. Also, any built-in job scheduling system on a cluster front-end would have to be bypassed.

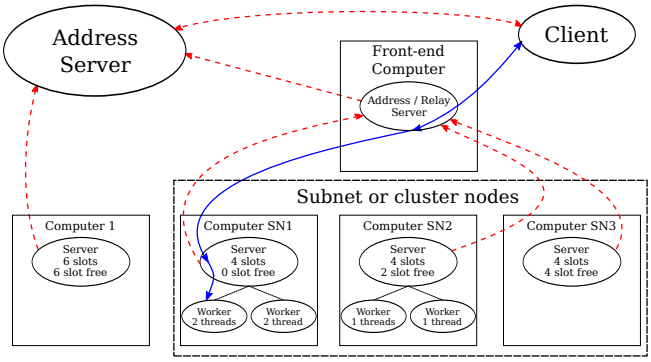

Figure 4. If the network contains subnets or cluster nodes that are not available through port forwarding, an address server instance with access to both networks must be used as a relay.

2.3 Continuous Performance Monitoring

Before a job starts, all participating servers should have supplied benchmark results indicating their performance. However, as the intended target is a work-place network of computers, the performance of the ongoing simulation work must be continuously monitored. Computers may at any time unexpectedly disconnect or slow down due to external use. The work may then need to be rescheduled. For this reason, the scheduler or clients can request simulation progress from each worker to determine if one is significantly lagging behind the others.

A computer with a fast benchmark score may still be unsuitable due to external use. An extension would therefore be to monitor the average CPU utilization. It is, however, difficult to estimate for how long demanding external work will continue.

2.4 Implementation

The framework has been implemented in C++ to work together with the Hopsan (Eriksson et al., 2010) simulation tool. The framework is centred around the message passing library ZeroMQ (Hintjens, 2013). This library can be used for both inter-thread, inter-process and inter-machine communication. It supports a variety of protocols and language bindings are available for may common languages. The library also has built in support for useful communications patterns, such as the *request-reply* pattern. In this mode incoming requests are automatically queued on a socket until the previous requests have received their reply. A *router* type socket also facilitates asynchronous I/O on the same port. This is useful when setting up a relay server for passing messages between networks.

All network components communicate over TCP/IP. ZeroMQ makes it possible to start servers and clients in any order, making it easy to resume communication when they come and go. While the network components are all written in C++, the use of ZeroMQ makes it possible to include software written in other languages. ZeroMQ can be built with no external dependencies which simplifies implementation and deployment. It can also be extended with support for authentication and encryption.

It is assumed that the optimization algorithms are implemented by the client, in this case the Hopsan simulation tool. The framework implementation consists of:

- The simulation service provider (server) application
- A worker process for the Hopsan simulation core
- The address server, for tracking the available providers on the network
- A client library, simplifying communication with the servers and workers
- The job-scheduler module as a library, currently tightly coupled to the Hopsan simulation tool

The messages exchanged in the framework are currently specific to Hopsan, but should be generalizable for similar tools. Examples are:

- Simulation slot request / reservation, including preferred number of threads for model parallelism
- Sending simulation models and assets like data input files, sub-model libraries and FMUs used by a model
- Simulation control; start, stop and step times, abort or single step instructions for real-time control
- Set and retrieve model parameters
- Status from servers; the number of total/open slots, what services they provide, benchmark speed
- Status from workers; simulation progress
- Retrieving simulation results, variable names, units and values
- Real-time streaming of simulation input and output (if possible)

3 Parallel Optimization Methods

Parallel optimization algorithms are required to evaluate the scheduling algorithm. Two common families of optimization algorithms have been investigated: direct search and population based methods. Only non-gradient based methods have been included. These are more efficient in solving non-linear problems, and can be applied also on problems which are not differentiable.

DOI 10.3384/ecp15119187

3.1 Direct Search Methods

Direct search optimization (Hooke and Jeeves, 1961) refers to methods where the parameter space is searched by a number of trial points. At each iteration, an algorithm is used to move one or more points depending on the location of the best known point. This is useful for discrete or non-continuous models, where the gradient cannot easily be obtained. Advantages typically include few evaluations and simple parameterization. On the downside, the possibilities for parallelization are often limited.

The most well-known direct search method is the Nelder-Mead Simplex algorithm for unconstrained problems (Nelder and Mead, 1965). It searches the n-dimensional parameter space with $k = n + 1$ points called the "simplex". The worst point is reflected through the centroid of the remaining points. Depending on the outcome, the simplex can be expanded, contracted or reduced.

A related method for constrained problems is the Complex method (Box, 1965). In contrast to the simplex method it uses at least $k = n + 2$ points and a reflection factor $\alpha > 1$. This causes a continuous enlargement of the complex. If the reflected point is still the worst, it is retracted iteratively towards the centroid, which compensates for the enlargement. This reduces the risk of the complex collapsing into a subspace when a constraint is reached. Retractions can also be weighted towards the best known point, to prevent the complex from collapsing into the centroid (Guin, 1968). The method can be further improved by adding a random factor and a forgetting factor, commonly referred to as the Complex-RF method (Krus and Ölvander, 2003).

Even though the algorithm is sequential, it can be parallelized with minor modifications. Parallel implementations of the Simplex algorithm has been conducted by (Dennis and Torczon, 1991) using multi-directional search and by (Lee and Wiswall, 2007) by reflecting multiple points at each iteration. Another possibility is to use multiple reflection factors, or to use task prediction methods where possible future evaluations are predicted by assuming a certain outcome of the current step. Multiple retraction steps towards the centroid can also be evaluated in parallel. Here methods using task prediction, multi-retraction and a combination of the first two methods have been tested.

With the task prediction method, candidates are generated iteratively by assuming the outcome of the previous reflection. Figure 5 illustrates the method. First the worst point (x_3) is reflected through the centroid of the other points. It is now assumed that the new point x_3' is better than the second worst point x_1. Therefore x_1 can be reflected through x_2, x_4 and x_3'. This process is repeated for all four original points. Should the number of simulation slots exceed the number of points, additional candidates are generated by iteratively moving the first reflected point towards the centroid. All candidates are then evaluated in parallel. If results show that one of the reflected candidates is still the worst, the remaining reflected points are discarded. Moving the reflected points iteratively towards the centroid can then easily be performed in parallel. If it is the first reflected point that shall be moved, the additional candidates can be used to speed up this process further. The iteration will continue until a point that is better than the previously worst point is found.

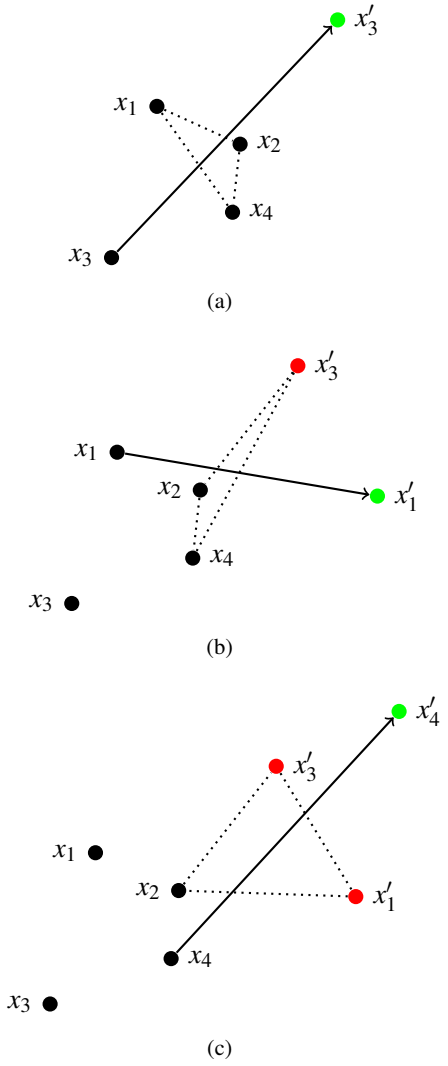

Figure 5. Task-prediction parallel Complex-RF algorithm. By assuming an outcome of a reflection, the next point can be reflected in advance.

Parallelizing retraction steps is trivial, since each step is independent of the previous ones. The method is illustrated by Figure 6. An unlimited number of steps can be computed in parallel. Once a retraction point that is no longer the worst point in the complex is found, all remaining points are discarded.

Results showed that the parallel efficiency of both methods decrease with an increasing number of processing units. Higher speed-up can be achieved by using both methods together and letting them share the available CPUs. In these experiments each method use half

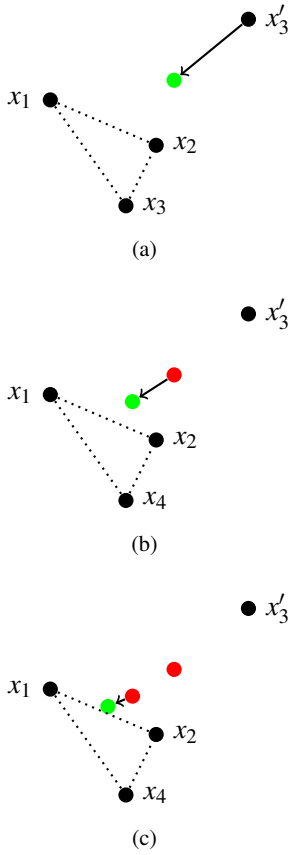

Figure 6. When retracting towards the centroid using the Complex-RF method, multiple steps can be evaluated in parallel.

of the CPUs. Obviously other methods may provide better results, but this has not been considered further in this paper.

The direct search methods were tested on three optimization problems; the sphere function (Equation 1), the Rosenbrock function (Equation 2) and a simulation model of a hydraulic position servo, shown in Figure 7. In the latter, the objective is to minimize the error between reference position and actual position while also maximizing the total energy efficiency of the system. Five parameters are used; proportional and integral control parameters K_p and K_i, piston areas A_1 and A_2 and pump displacement D_p.

$$\begin{aligned}
\text{minimize} \quad & \mathbf{F}(\mathbf{x}) = (x_{error}(\mathbf{x}), \eta_{tot}(\mathbf{x}))^T \\
& \mathbf{x} = (K_p, K_i, A_1, A_2, D_p)^T \\
\text{subject to} \quad & 0 \leq K_p \leq 1 \times 10^{-2} \\
& 0 \leq K_i \leq 1 \times 10^{-3} \\
& 1 \times 10^{-5} \leq A_1 \leq 1 \times 10^{-2} \\
& 1 \times 10^{-5} \leq A_2 \leq 1 \times 10^{-2} \\
& 1 \times 10^{-6} \leq D_p \leq 1 \times 10^{-3}
\end{aligned}$$

The three models were optimized with each algorithm 100 times for each number of simulation slots, ranging from 1 to 8. Optimization parameters as suggested by (Box, 1965) and (Krus and Ölvander, 2003) are used, see Table 1. Results are shown in Figures 8, 9 and 10.

k	$2n$	r_{fac}	0.1
α	1.3	γ	0.3

Table 1. Number of points, reflection factor, randomization factor and forgetting factor used in experiments.

$$F(x_1, x_2) = x_1^2 + x_2^2 \tag{1}$$

$$F(x_1, x_2) = 100(x_2 + x_1^2)^2 + (x_1 - 1)^2 \tag{2}$$

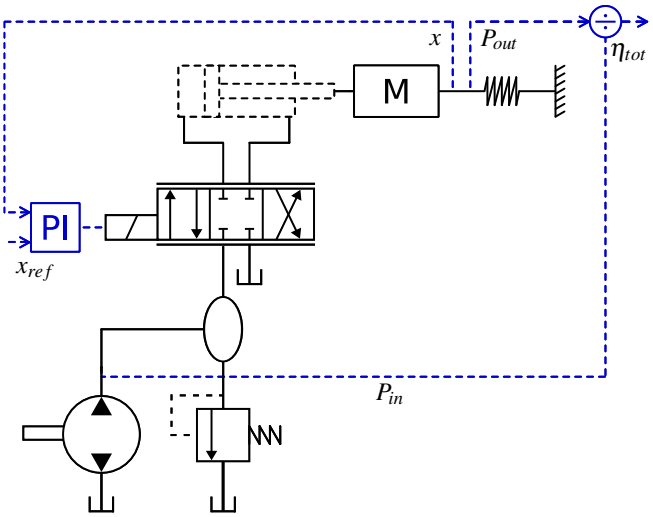

Figure 7. A model of a hydraulic position servo used to evaluate the parallel optimization algorithms.

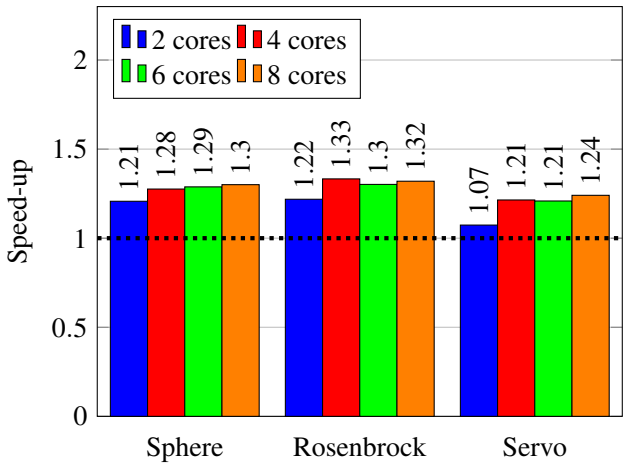

Figure 8. Algorithm speed-up with the task prediction method.

3.2 Population Based Methods

Population based methods is a family of optimization methods where a population of independent points are

DOI 10.3384/ecp15119187

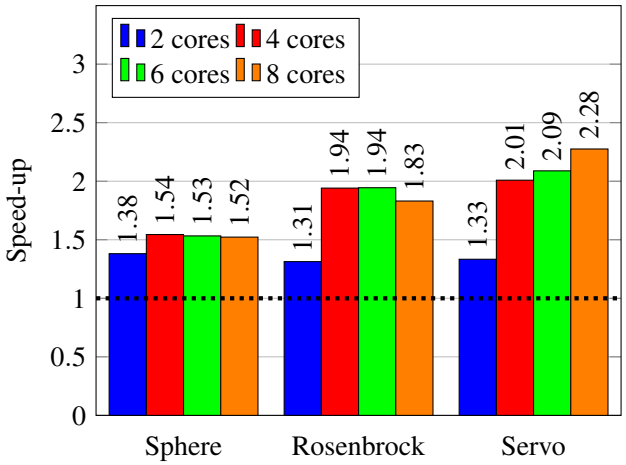

Figure 9. Algorithm speed-up with the multi-retraction method.

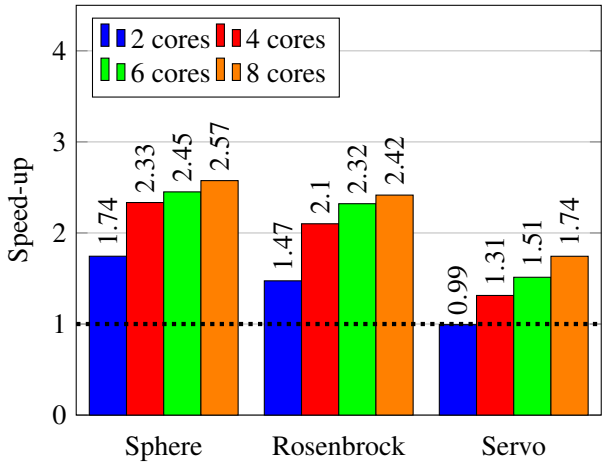

Figure 10. Algorithm speed-up with a combination of task prediction and multi-retraction. This is by far the most efficient method.

used to search the parameter space. At each iteration, all points are evaluated independently of each other. The outcome is then used to generate a new population, which is used in the next iteration. Due to the independent evaluations, population based methods are naturally parallel. Another benefit is that they are intuitive, since they often resemble physical, social or biological phenomenas. Examples of population based methods are Differential Evolution (DE) (Storn and Price, 1997), Genetic Algorithms (GAs) (Goldberg, 1989) and Particle Swarm Optimization (PSO) (Kennedy and Eberhart, 1995).

In this paper a PSO algorithm has been used as an example. Results can, however, be generalized to any naturally parallel optimization method. With PSO, each point (denoted "particle") has a position and a velocity vector, see Figure 11. At each iteration the velocity is changed depending on the particle's own best known position and the best known position in the swarm. This resembles gravitational pull. Each particle also has an inertial weight, that prevents rapid change in velocity. Particles are then moved according to the velocity before next iteration. Parallel speed-up can be assumed to be linear, and is only limited by the number of particles in the swarm. It should be pointed out that even though population based methods have higher speed-up than direct search methods, they also require significantly more iterations. Thus, they are not necessarily more efficient than direct search methods.

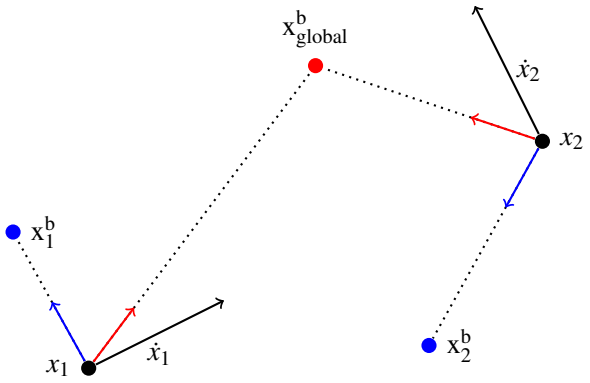

Figure 11. Population-based methods, such as particle swarm optimization, evaluate all points independently and are naturally parallel.

4 Work Scheduling

Efficient scheduling of the workload will have great impact on execution times. This task is complicated for several reasons. The number of available computers and/or processor cores may be limited. Computers may also have different performance or different number of cores. Furthermore, several clients may want to run different optimization jobs at the same time. The proposed solution is to estimate the speed-up factor for various work distributions on available resources.

Optimizations can be parallelized either on algorithm-level or model-level. Speed-up from parallelism will thus depend on both the algorithm and the model. A third alternative is to run several optimizations in parallel and pick the best result (algorithm coarse-grained or job-level parallelism). This has not been considered further in this paper.

Speed-up of the algorithm comes from either reducing the total number of sequential evaluations, or from reducing the number of needed iterations by searching the parameter space more efficiently. For this reason the *equivalent number of iterations with parallelism* $n_{eval,p}$ is defined:

$$n_{eval,p} = \frac{n_{eval}}{SU_a(p_a)} \tag{3}$$

At model-level, the speed-up is defined as the ratio between sequential and parallel execution time. The number of processor cores will determine the magnitude of the speed-up. In most cases it can be assumed that models

cannot be parallelized over several computers. Communication latency over the network induces overhead, which makes this efficient only for large models.

$$t_{sim} = \frac{t_{base}}{SU_m(p_m)} \tag{4}$$

The total execution time for an optimization can be calculated by combining Equations 3 and 4. At each iteration the algorithm will have to wait for the slowest simulation to finish. Thus, the maximum simulation time must be used according to Equation 5. The communication time for sending results over the network may also matter, but is assumed to be negligible for large simulation problems.

$$t_{opt} = n_{eval,p} \max_{0<i<k} t_{sim,i} + t_{comm} \tag{5}$$

For a homogeneous computer network, where all computers have the same computational speed and number of cores, Equation 5 can be simplified into Equation 6:

$$t_{opt} = n_{eval,p} t_{sim} \tag{6}$$

Finding the optimal task distribution is a combinatorial optimization problem. It can be expressed either as minimizing the execution time or as maximizing the speed-up. For heterogeneous systems, where computers have different performance and number of cores, it can be formulated as follows:

$$\begin{aligned} \text{maximize} \quad & SU(p_a, \mathbf{p_m}) = SU_a(p_a) \min_{0<i<k} SU_{m,i}(p_{m,i}) \\ \text{subject to} \quad & p_a \frac{1}{k} \sum_{i=0}^{k} p_{m,i} \leq \sum_{i=0}^{n_p} (n_{c,i}) \\ & p_{m,i} \leq n_{c_i} \text{ for } i = 0, ..., n_p \end{aligned}$$

Again, the formulation can be simplified for homogeneous systems:

$$\begin{aligned} \text{maximize} \quad & SU(p_a, p_m) = SU_a(p_a) SU_m(p_m) \\ \text{subject to} \quad & p_a p_m \leq n_p n_c \\ & p_m \leq n_c \end{aligned}$$

For a homogeneous system, it is sufficient to loop through different values for p_m, and calculate the largest possible corresponding p_a. The total speed-up can then be computed as the product of $SU_a(p_a)$ and $SU_m(p_m)$. Pseudo code for this is shown in Listing 1.

Listing 1. A sheduling algorithm that computes the optimal combination of algorithm and model parallelism (p_a^* and p_m^*) for homogeneous computer networks.

```
SU* = 0
for (pm=1; pm ≤ nc; ++pm) {
  pa = np floor(nc/pm)
  SU = SUa(pa)SUm(pm)
  if (SU>SU*) {
    SU* = SU
    p*m = pm
    p*a = pA
  }
}
```

Much of this code can be re-used for heterogeneous systems. Instead of using the same p_m for each computer, different maximum values (p_m^{max}) are tested. This limits the maximum number of cores each model can use. The number of parallel models that should run on each computer is then calculated. The sum of these gives the total value for p_a. Finally, SU_m is computed as the minimum speed-up from all computers. Pseudo-code for heterogeneous scheduling is shown in Listing 2.

Listing 2. For heterogeneous computer networks an approximate scheduling can be computed by limiting maximum level of model parallelism (p_m^{max}).

```
SU* = 0
for (p^max_m=1; p^max_m ≤ n^max_c; ++pm) {
  pa = floor(Σ_{i=0}^{np} nc,i / min(p^max_m, nc,i))
  SU = SUa(pa)SUm(pm)
  if (SU>SU*) {
    SU* = SU
    p*m = pm
    p*a = pA
  }
}
```

5 Results

Two experiments were conducted, one theoretical and one practical. First, the scheduling algorithm was numerically verified by letting it calculate probable speed-up for two test models on a fictional computer network. Subsequently, the scheduler was implemented and tested by running optimizations on real hardware.

5.1 Numerical Verification

The scheduling algorithm was numerically verified by using two example models; the position servo model mentioned previously, and a model of a hydraulic mining rock drill. The latter is a large model with good opportunities for model-level parallelism. Speed-up on a quad-core processor was measured to be 2.60. The model is provided by industrial partners, and is used in real product development. For the first model, however, model-level parallelism is not beneficial due to the low number of submodels. Execution time appear to be independent of the number of cores.

As a theoretical example the scheduler was executed for four homogeneous computers with four cores each. By using pre-defined speed-up factors, the optimal work distribution can be obtained from a look-up table. All three optimization algorithms mentioned in Section 3 were examined. The PSO algorithm was assumed to use 16 particles. Speed-up of the algorithms was assumed to be the

DOI 10.3384/ecp15119187

same for the rock drill as for the position servo. Results are shown in Table 2. As expected, the rock drill is calculated to achieve the best performance when using the maximum number of cores at model-level. The position servo, however, benefits more from a high degree of algorithm parallelism. When using PSO, parallelism at algorithm level is always more beneficial.

			Position Servo				Rock Drill			
		p_m	1	2	3	4	1	2	3	4
	p_a	SU_m / SU_a	1.00	1.00	1.04	0.96	1.00	1.92	2.23	2.60
Complex-RFP	1	1.00	1.00	1.00	1.04	0.96	1.00	1.92	2.23	2.52
	2	0.99	0.99	0.99	1.03	0.95	0.99	1.90	2.21	2.49
	3	1.19	1.19	1.19	1.24	1.14	1.19	2.28	2.65	3.09
	4	1.31	1.31	1.31	1.36	1.26	1.31	2.52	2.92	**3.41**
	5	1.42	1.42	1.42	-	-	1.36	2.73	-	-
	6	1.51	1.51	1.51	-	-	1.45	2.90	-	-
	7	1.67	1.67	1.67	-	-	1.60	3.21	-	-
	8	1.74	1.74	**1.74**	-	-	1.67	3.34	-	-
PSO (with 16 particles)	1	1.00	1.00	1.00	1.04	0.96	1.00	1.92	2.23	2.60
	2	2.00	2.00	2.00	2.09	1.92	2.00	3.83	4.46	5.2
	3	2.00	2.00	2.00	2.09	1.92	2.00	3.83	4.46	5.2
	4	4.00	4.00	4.00	4.17	3.84	4.00	7.66	8.91	10.4
	5	4.00	4.00	4.00	-	-	4.00	7.66	-	-
	6	5.33	5.33	5.33	-	-	5.33	10.23	-	-
	7	5.33	5.33	5.33	-	-	5.33	10.23	-	-
	8	8.00	8.00	8.00	-	-	8.00	15.33	-	-
	9	8.00	8.00	-	-	-	8.00	-	-	-
	10	8.00	8.00	-	-	-	8.00	-	-	-
	11	8.00	8.00	-	-	-	8.00	-	-	-
	12	8.00	8.00	-	-	-	8.00	-	-	-
	13	8.00	8.00	-	-	-	8.00	-	-	-
	14	8.00	8.00	-	-	-	8.00	-	-	-
	15	8.00	8.00	-	-	-	8.00	-	-	-
	16	16.00	**16.00**	-	-	-	**16.00**	-	-	-

Table 2. Speed-up factor as a function of work distribution for the combined Complex-RFP and PSO algorithms. Optimal distributions are shown in red.

5.2 Load Balancing Experiments

To verify the load balancing and rescheduling capabilities of the framework, a network of eight dual-core computers with equal performance was used. The experiment model was a slightly modified position servo model with a dual-core model-level speed up 1.12. Model-level parallelism is achieved through the use of the transmission line element method (Krus et al., 1990) and multi threading (Braun et al., 2011). The model is optimized with both the combined Complex-RFP and the PSO methods. Algorithm speed-up for the Complex-RFP algorithm is taken from Table 2. While the PSO algorithm is assumed to have linear speed-up in itself, in practice this depends on the number of particles used and the number of available computers according to Equation 7. The speed-up is the ratio between the sequential execution time of all models and the longest queue required on any of the simulation servers during parallel execution. The initial algorithm-level parallelism was set to six in both cases, leaving two computers unallocated.

$$SU_{pso,actual} = \frac{p_a}{\text{ceil}\left(\frac{p_a}{\min\left(n_p \text{ floor}\left(\frac{n_c}{p_m}\right), p_a\right)}\right)} \quad (7)$$

Figure 12 shows the total iteration time with the Complex-RFP method including evaluation and transfer of results over the network. One participating computer after another becomes overloaded by external work. Before *c*, the load can be shifted to the free computers and no slowdown occur. After *c*, since the algorithm speed-up is low for this method, the algorithm is reinitialized to use fewer parallel models. At *f* it finally becomes more beneficial to increase the number of parallel models again in comparison to maintaining dual-core model-level parallelism. For a model this small and fast the transfer of result data cannot be neglected. The results show between *c* and *f* that the total time is decreasing slightly with fewer parallel models as less data needs to be transferred.

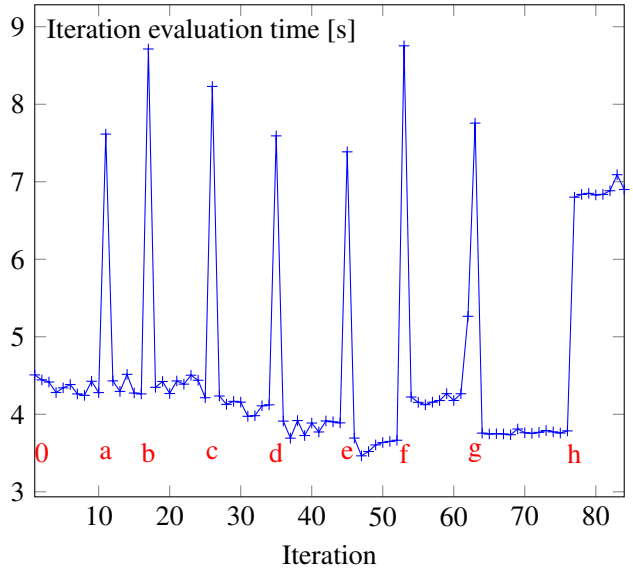

Figure 12. Evaluation time for one iteration using the combined Complex-RFP algorithm, beginning with six parallel evaluations. The letters *a*–*h* each represent one additional network computer becoming overloaded. Load balancing or parallelization reduction is shown in Table 3.

Figure 13 shows the same experiment for the PSO algorithm. Since it is naturally parallel, the number of particles (parallel models) is kept fixed at six. The speed-up in this case comes purely from the possibility of executing models in parallel. When computers become unavailable the scheduler will try to balance the load so that queues are avoided. Since it is unlikely that a model-level speed-up larger then the number of cores used can be achieved, the method always favors the highest possible algorithm parallelism. At *c*, free computers are no longer available and the scheduler switches to single-core simulation, which is slightly slower but allows to maintain parallel execution. At *f*, the scheduler is forced to begin queuing models for execution and the evaluation time is doubled.

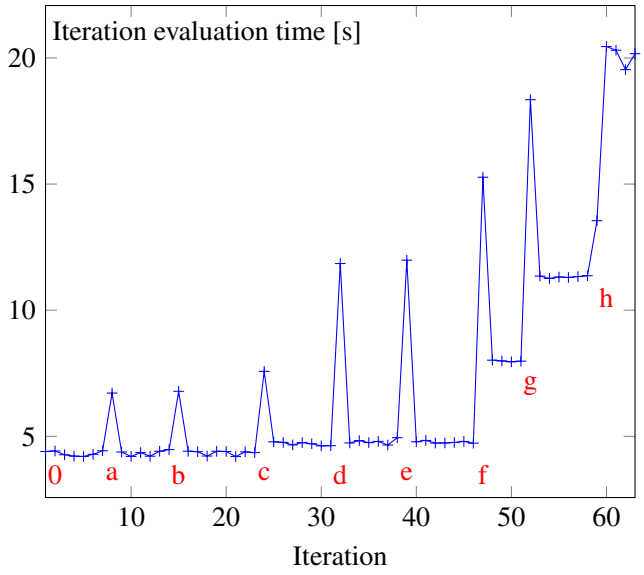

Figure 13. Evaluation time for one iteration using the PSO algorithm with six parallel models. The letters *a–h* each represent one additional network computer becoming overloaded. Load balancing results are shown in Table 3.

In both figures, the spikes represent approximately double execution time, since when a server is lost the current iteration is restarted. At *h*, only one computer remains and when it becomes heavily loaded no better alternatives exist. The chosen parallelism for each rescheduling is given in Table 3.

		0	a	b	c	d	e	f	g	h
$n_p \times n_c$		16	14	12	10	8	6	4	2	1
C-RFP	p_m	2	2	2	2	2	2	1	1	1
	p_a	6	6	6	5	4	3	4	2	2
PSO	p_m	2	2	2	1	1	1	1	1	1
	p_a	6	6	6	6	6	6	3	2	2

Table 3. The combination of model-level and algorithm-level parallelism that gives the best speed-up depending on availability of computers on the network. Each column represents a point in Figures 12 and 13.

6 Conclusions

A framework for work scheduling of multi-level parallel optimizations has been developed. The scheduler takes the number of available processor cores on each computer into account. Therefore, parallelism on both model-level and algorithm-level can be used to maximize performance. An extension could be to also support solver-level parallelism, for example parallel matrix operations. This could be solved using GPU cards, which would add an additional level to the scheduler.

For homogeneous computer networks, an optimal scheduling can be found by testing all combinations. Heterogeneous networks, however, require heuristic scheduling methods. Thus, an optimal solution cannot be guaranteed.

The speed-up from parallel optimization algorithms depend to a great extent on the model being optimized. Especially, the combination of algorithm and model has a great influence on performance improvement. Thus, different algorithms are efficient for different models. Some knowledge of the model properties are therefore required.

Experiments show that the framework is able to deal with computers on the network suddenly becoming overloaded. The scheduler is able to re-balance the computation load over the remaining resources.

While the implementation is specific for the Hopsan simulation environment, the methodology can be generalized to support other tools. The FMI standard makes it possible to run optimizations with FMUs from other programs embedded in a Hopsan model. Extending the implementation to allow simulation of the FMUs directly is an important continuation of this work. This will, however, require each FMU to contain information of its performance and parallelizability. Including such information in the FMI standard could be one possibility.

Nomenclature

k	Number of points
n	Number of parameters
α	Reflection factor
r_{fac}	Randomization factor
γ	Forgetting factor
p	Degree of parallelization
n_{eval}	Number of evaluations
$n_{eval,p}$	Equivalent evaluations with parallelism
p_a	Degree of parallelization of algorithm
p_m	Degree of parallelization of model
SU_a	Speed-up of algorithm
SU_m	Speed-up of model
SU_{opt}	Speed-up of optimization
t_{sim}	Simulation time
t_{base}	Sequential simulation time
t_{opt}	Optimization time
n_c	Number of processor cores
n_p	Number of processors or computers

References

David P Anderson. Public computing: Reconnecting people to science. In *Conference on Shared Knowledge and the Web*, pages 17–19, 2003.

T. Blochwitz, M. Otter, M. Arnold, C. Bausch, C. Clauß, H. Elmqvist, A. Junghanns, J. Mauss, M. Monteiro, T. Neidhold, D. Neumerkel, H. Olsson, J.-V. Peetz, and S. Wolf. The functional mockup interface for tool independent exchange of simulation models. In *8th International Modelica Conference 2011*, Como, Italy, September 2009.

M. J. Box. A new method of constrained optimization and a comparison with other methods. *The Computer Journal*, 8 (1):42–52, 1965. doi:10.1093/comjnl/8.1.42.

Robert Braun, Peter Nordin, Björn Eriksson, and Petter Krus. High Performance System Simulation Using Multiple Processor Cores. In *The Twelfth Scandinavian International Conference On Fluid Power*, Tampere, Finland, May 2011.

John E Dennis, Jr and Virginia Torczon. Direct search methods on parallel machines. *SIAM Journal on Optimization*, 1(4): 448–474, 1991. doi:10.1137/0801027.

MS Eldred, WE Hart, BD Schimel, and BG van Bloemen Waanders. Multilevel parallelism for optimization on MP computers: Theory and experiment. In *Proc. 8th AIAA/USAF/-NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization, number AIAA-2000-4818, Long Beach, CA*, volume 292, pages 294–296, 2000. doi:10.2514/6.2000-4818.

B. Eriksson, P. Nordin, and P. Krus. Hopsan NG, A C++ Implementation Using The TLM Simulation Technique. In *The 51st Conference On Simulation And Modelling*, Oulu, Finland, 2010. URL `http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-60644`.

Robert Fourer, Jun Ma, and Kipp Martin. Optimization services: A framework for distributed optimization. *Operations Research*, 58(6):1624–1636, 2010. doi:10.1287/opre.1100.0880.

Björn Gehlsen and Bernd Page. A framework for distributed simulation optimization. In *Proceedings of the 33nd conference on Winter simulation*, pages 508–514. IEEE Computer Society, 2001. doi:10.1109/WSC.2001.977331.

David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989. ISBN 0201157675.

J. A. Guin. Modification of the complex method of constrained optimization. *Computer Journal*, 10(4):416, 1968. ISSN 00104620. doi:10.1093/comjnl/10.4.416.

Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. "O'Reilly Media, Inc.", 2013.

Robert Hooke and T. A. Jeeves. "Direct Search" Solution of Numerical and Statistical Problems. *J. ACM*, 8(2):212–229, April 1961. ISSN 0004-5411. doi:10.1145/321062.321069.

J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948 vol.4, 1995.

P. Krus, A. Jansson, J-O. Palmberg, and K. Weddfelt. Distributed simulation of hydromechanical systems. In *The Third Bath International Fluid Power Workshop*, Bath, England, 1990.

Petter Krus and Johan Ölvander. Optimizing optimization for design optimization. In *Design Engineering Technical Conferences and Computers and Information in Engineering Conference,2003*. ASME Press, 2003. doi:10.1115/DETC2003/DAC-48803.

Donghoon Lee and Matthew Wiswall. A parallel implementation of the simplex function minimization routine. *Computational Economics*, 30(2):171–187, 2007. doi:10.1007/s10614-007-9094-2.

J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965. doi:10.1093/comjnl/7.4.308.

N. Sadashiv and S.M.D. Kumar. Cluster, grid and cloud computing: A detailed comparison. In *Computer Science Education (ICCSE), 2011 6th International Conference on*, pages 477–482, Aug 2011. doi:10.1109/ICCSE.2011.6028683.

Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997. doi:10.1023/A:1008202821328.

Enver Yücesan, Yuh-Chuyn Luo, Chun-Hung Chen, and Insup Lee. Distributed web-based simulation experiments for optimization. *Simulation Practice and Theory*, 9(1):73–90, 2001. doi:10.1016/S0928-4869(01)00037-4.