information

Article

Tree-Like Distributed Computation Environment with Shapp Library

Tomasz Gałecki and Wiktor Bohdan Daszczuk *

Institute of Computer Science, Warsaw University of Technology, 00-665 Warsaw, Poland; tomasz.galecki.stud@pw.edu.pl
* Correspondence: wbd@ii.pw.edu.pl; Tel.: +48-22-234-78-12

Received: 30 January 2020; Accepted: 1 March 2020; Published: 3 March 2020

Abstract: Despite the rapidly growing computing power of computers, it is often insufficient to perform mass calculations in a short time, for example, simulation of systems for various sets of parameters, the searching of huge state spaces, optimization using ant or genetic algorithms, machine learning, etc. One can solve the problem of a lack of computing power through workload management systems used in local networks in order to use the free computing power of servers and workstations. This article proposes raising such a system to a higher level of abstraction: The use in the .NET environment of a new *Shapp* library that allows remote task execution using fork-like operations from Portable Operating System Interface for UNIX (POSIX) systems. The library distributes the task code, sending static data on which task force is working, and individualizing tasks. In addition, a convenient way of communicating distributed tasks running hierarchically in the *Shapp* library was proposed to better manage the execution of these tasks. Many different task group architectures are possible; we focus on tree-like calculations that are suitable for many problems where the range of possible parallelism increases as the calculations progress.

Keywords: workload management; remote fork; distributed computations; task group communication

1. Introduction

Calculations performed in modern information systems are so time-consuming that their performance on a single processor, even a very fast one, does not allow one to achieve results in a reasonable time. Examples include mass simulations that find the best possible set of parameter values for a certain system, genetic algorithms, or searching for the tree of possible transformations.

Such calculations can be carried out in supercomputer environments [1], in which they are performed in ways that are extremely quick, but also extremely expensive. Instead, the computing resources of personal computers and servers in the local network or student laboratory environment can be used. If fragments of calculations are independent of each other, they can simply be sent to remote computers, and then their results can be collected. For example, in the simulation analysis of a space of solutions in which individual simulations are completely independent of each other, model parameters are sent to the computers and the results are collected. This is the characteristics of research on the efficiency of autonomous urban transport carried out with the help of the *Feniks* simulator [2] by the authors. This has been a primary motivation to begin the work on *Shapp* library.

Otherwise, in the dynamic programming approach to problem solving, e.g., as in chess play, the next move (or possible sequence) depends entirely on the current situation on the chessboard. However, the same situation on the chessboard may occur in different branches of the move tree, which may lead to unnecessary parallel analysis of the same subtree of moves (the tree becomes a more general graph, perhaps even cyclical). Therefore, it would also be useful to be able to communicate between tasks to eliminate such situations.

Even closer communication between tasks is required in the case of ant colony algorithms, where the probability of the ant choosing specific moves depends on the behavior of other ants [3]. In turn, the non-exhaustive model checking algorithm in the *DedAn* environment consists of making hypotheses by one task and verification of these hypotheses by another one, which requires their cooperation in sending the parameters of the hypothesis, which is different each time (it starts from a different configuration of the verified system, but the configuration history is also needed for cutting the loops in the search graph) [4].

Ideally, in a distributed computation system a process can perform an operation similar to the fork operation known from POSIX systems (in which a faithful copy of the process is created), and branched processes can communicate [5]. The fork operation transfers to the child process an exact copy of the parent's calculation state, and sending messages and exit codes between processes allows for controlling further progress of the calculations.

Such a situation requires significant data transfers in a distributed environment where there is no shared memory in which it is easy to duplicate the process memory area, in which both the state of global variables and the dynamic state of the execution stack are stored. Communication between processes in a task group is also possible in various ways, such as pipes, remotely shared data blocks, or exit codes. In this article, we propose a solution in the form of a *Shapp* library using the HTCondor workload management system [6–11]. Our solution can be called a distributed fork (or remote fork), in which the process can perform cloning, transferring the state of calculations in the form of files containing global variables and dynamic state of calculations (in the *DedAn* program—the model of the verified system and the search path). An important difference compared to the local fork operation is that the child process starts from the beginning of the program (main function) and not from the branch point, because the execution stack is not copied. For this price, we get the ability to start child processes in a heterogeneous environment, for example, Windows and Linux. In addition, the ability to send messages between distributed program copies and to capture the exit code has been implemented.

The *Shapp* library was created to meet computational needs in our research projects. The first was the mentioned *Feniks* simulator, in which it was necessary to carry out a large number of hours-lasting simulations, which run concurrently and independently of each other, and to collect the results of all these simulations. The second challenge was the non-exhaustible state space search algorithm in the *DedAn* temporal verifier. Here, the structure of processes is more complicated because some processes start others, being clones of their parents or performing other functions, and there is a need for communication between processes and control of the execution of child processes by their parents. This complicated relationship structure is described later in the article.

In Section 2 we present various approaches to distributed workload management. Section 3 recalls the development of the *Shapp* library. Section 4 covers its application to non-exhaustive model checking in the *DedAn* system. The details of the *Shapp* library are described in Section 5. Section 6 presents communication aspects within a distributed task group. Section 7 shows an example of *Shapp* application to ACO (ant colony optimization) computation. Section 8 concerns future work.

2. Related Work

An overview on the situation in the market of distributed computing share was discussed in more detail in the article about *Shapp* library basics [12]. Only the essentials will be described here, to provide general context. Especially, the context behind the decision made towards a base system for the *Shapp* library.

2.1. Distributed Workload Management Systems

The distributed workload management systems are the implementations of the idea of mass distributed computing. Computational units are individual programs that are repeatedly executed for different sets of input data [13]. Examples of the use of distributed mass processing systems include:

- BOINC [14]—a system based on the use of computing power provided by Internet users. Example subprojects are: ATLAS@Home [15]—simulation of the ATLAS project (in LHC), Asteroids@home [16]—broadening the knowledge of asteroids, Big and Ugly Rendering Project (BURP) [17]—a non-commercial and open project focusing on the rendering of three-dimensional scenes in the cloud.
- Comcute [18]—the system dealing, among other things, with the genetic testing of a radical settlement, techniques and processing of images.

The majority of systems use already existing stable middleware. This middleware is a basis for creation of distributed computation solutions. Typical middleware is responsible for [19]:

- Administration of resources (computers in the network).
- Management of a task queue.
- Support for task prioritization.
- Matchmaking of tasks to be carried out, which fulfill specific requirements.
- Providing an application programming interface (API) that simplifies the management of the entire system.

These types of tasks appear in every distributed computing system. For example, the BOINC project [20], which is a framework program for computations on machines made available by Internet users, is based on the HTCondor system [6]. Individual systems are distinguished by their configuration (in case they are based on the same software).

2.2. Types of Distributed Processing Systems

Distributed processing systems can be divided into groups considering different features. In the literature, two ways of classification were observed by the authors:

- Structural classification [21].
- Functional classification [22].

2.2.1. Structural Classification

The first approach to distributed workload management systems is structural classification [21]. Its main concern is the architecture behind hardware implementation of the computing pool.

Computer cluster. A computer cluster is a group of computers connected to each other so as to allow each system to be transparent. In contrast to network computing, each network node performs the same task. Computer clusters typically carry out high-performance tasks. The middleware used in clusters depends on the purpose of the given system:

- *Distcc* [23]—automatized to run distributed compilation.
- Linux Virtual Server [24]—software for building general purpose clusters. It is based on several systems with greater specialization, e.g., UltraMonkey [25]—a process-based process system for supporting a computing pool over the internet/intranet, Keepalived [26]—a distributed process-based system simplifying the use of Linux Virtual Server to ensure greater reliability and high availability of the calculation pool.

Network processing (grid computing). Network processing uses highly distributed, heterogeneous computer units. From the point of view of the user, this idea uses open protocols and general purpose interfaces in order to build a virtual and powerful computing environment from a network of connected different nodes. The BOINC project [20] mentioned above is an example of this. Network processing most often accomplishes tasks from the high-throughput processing class (described in Section 2.2.2). Middleware implementations are similar, they differ in additional functionalities that facilitate the use of given systems for specific types of tasks. Common examples are: Eucalyptus [27]—a paid tool for the integration and use of Amazon Web Services, HTCondor [6]—a free tool with support for heterogeneous network systems.

2.2.2. Functional Classification

Second approach is functional classification [22]. There are three types of distributed computing systems due to the type of processing: High-throughput computing [28], high-performance computing [29] and multi-task computing [30]. However, the difference between high-performance and multi-task computing is not strict [22]. Nonetheless, each category has its own basic characteristics.

- High-throughput computing. High-speed processing focuses on supporting computing units that require chaining. This class operates on a macro scale—the processing time of individual computing units is calculated in hours or even days. Examples of the use of this type of system are: Transparent conducting materials discovery [31] or minimizing cascading blackout risk [32].
- High-performance computing. In high-performance processing computing units can be completely paralleled (pool computing). This class operates on a micro scale—the processing time of individual computing units (tasks) is counted in milliseconds. It applies to the computing clusters. Examples of the use of this type of system are: Development of results of collisions in Large Hadron Collider [33] or materials development [34].
- Multi-task computing. Multi-tasking is an intermediate solution between high-throughput and high-performance processing. Tasks can be both sequential and parallel. The process concentrates on cooperation with the file system. Examples of the use of this type of system are: High-performance, distributed databases [22] or search engines [30].

3. Development of the *Shapp* Library

3.1. The Choice of the Basic Environment

The *Shapp* library was initially designed for cooperation with *Feniks* personal rapid transit (PRT) network simulator [35]). The core characteristics then was time spent on single computational unit processing. In case of Feniks, a single simulation takes 2–3 h for a medium size model. The *Shapp* library was used for better research scaling, when multiple similar simulations have to be performed fast.

Experiments involve carrying out a significant number of simulations on the same model, with specific parameters changing step-wise within certain limits, or selecting parameter values from a specific set. This process, carried out manually, is monotonous, and worse, prone to errors when determining the values of parameters for a specific simulation and when collecting results. The *Feniks* simulator uses three methods to automate experiments:

- Input file generator with variable parameter values, with the same model and other parameter values not subject to change in the experiment.
- The result collector in an SQL database, allowing access to individual results and extracting trends resulting from simulation, finding extreme and optimal values.
- Automating parallel simulations using threads. This mechanism allows for running many simulations at the same time, but when the number of cores in the computer is exceeded, the simulations disturb each other because time is wasted for switching between processes. As a result of research on the parallelism of simulations, the *Shapp* library was developed, which allows for running a simulation pack in a distributed environment using a workload management tool.

To achieve this, the examined task was analyzed in terms of its properties:

- Type of processing. The first aspect that strongly limited the choice of available solutions is the type of processing required. Simulations of long-time tasks that cannot be paralleled meet class of high-performance processing systems.

- Execution environment. An important requirement is the ability to use currently available computing resources. The available computers are diverse, controlled by various operating systems (Linux, Windows). Only network processing supports such heterogeneous environment.
- Correct handling of .NET environment on Linux and Windows systems. The *Feniks* simulator is developed under the .NET environment, which is not fully supported on Linux systems, but it is possible to use the support of the Mono environment [23]. However, taking into consideration that most of the projects in our institute are developed in this technology, a library in the native language greatly improves its usability.
- License and cost. It was assumed that the used environment should be free, at least for academic use for an unlimited number of hosts.

Considering enumerated properties, five distributed workload management systems were selected. The comparison was described in Table 1.

Table 1. List of considered distributed workload management systems to use as a base for the *Shapp* library.

	Open Source	Cost	License	Operating Systems
Oracle Grid Engine	No	Paid	SISSL	Linux
TORQUE	No	Free	Private	Linux
Globus Toolkit	No	Free	Apache	Linux, MacOS
SLURM	Yes	Free	GNU	Linux
HTCondor	Yes	Free	Apache	Linux, MacOS, Windows

Given the requirements for a distributed computing system, HTCondor has proved to be the most accurate choice. Moreover, multiple publications have proven its usability and flexibility with real scenarios, for example:

- Integration with OneDrive cloud [36].
- Integration with OpenStack cloud aggregator [37].
- Integration with reproducible research data analysis platform (REANA) [38].
- On-demand provisioning of HEP compute resources [39].

CPU scavenging: HTCondor support for scavenging (in addition to simply using all available computer resources) helps in the utilization of free computing resources that may still have their functions as personal computers. Scavenging [40,41] is one of the functionalities offered by the HTCondor distributed computing system. It is based on the utilization of the computing power of the processor at times when the user does not use the computer. When the system detects that user input devices are inactive by the predefined time, the service proceeds to perform calculations. This functionality is optional. It can be replaced by, for example, reserving a number of cores for a user. HTCondor is not a new system, however, it is still successfully used in workload management. Numerous computation structures can be built upon HTCondor p2p architecture, which is extremely flexible compared to other solutions, and more fault tolerant. In the last few years, HTCondor has been integrated with Slurm clusters [8].

3.2. HTCondor Weak Points

After selecting the most suitable distributed processing system (HTCondor), we tried to apply it for *Feniks* simulations without additional support. The method of executing calculation units is difficult to apply when conducting multiple simulations of the same model with different sets of parameter values. This is due to the specificity of problems associated with research using the *Feniks* simulator. A good example is the experiment generator, with which the program performs a set of simulations, each with different parameter values but all on the same model. Sets of parameter values are generated using a neural network. After each epoch, results are collected, which are then used to modify the neural network coefficients. As a result, a new set of parameter values is generated and the cycle repeats. This would require an additional script to create batch-defining files that would only differ in input data.

There is also a different way to request the execution of the calculation units—application interface in Python 2. However, *HTCondor* does not provide support for any other programming language than Python, in particular, there is no solution supporting the C# language.

The last feature is a complete lack of hierarchy between the executed tasks. It is natural, as the HTCondor is a distributed workload management system, and it is designed to work with batch programs. However, it is a blocking point for any tree-like computations.

3.3. Fulfilling the Requirements

The biggest problem with the use of a distributed processing system is the impossibility of creation of the descendant batches in the simulator itself. It should be organized similarly to the *fork()* procedure known from POSIX class systems.

Such a solution allow creating simple batches (in this problem, the simulator application is put together with the simulation model and a set of simulation parameters) in the body of the calculation unit.

Additionally, the fork-like solution will add ability for tree-like processing structure applications.

4. Application of *Shapp* to Non-Exhaustive Model Checking

One of the most desirable attributes of distributed systems is their deadlock immunity. It is especially crucial in environments such as medical care equipment, aeronautics or space exploration, where reliability must be very strong [42]. Many of those real systems can be represented using structured and simplified models, like Petri nets [43], CSP [44] or SDL [45].

Such models can be then verified against total and partial deadlocks using known algorithms and tools. One of them is *DedAn* [46]—a deadlock analyzer developed in Institute of Computer Science, Warsaw University of Technology. An example problem is the analysis of potential deadlock in a fragment of Rome metro, shown in Figure 1 taken from Mazzanti et al. paper [47]. The result of the verification is a counter-example of a trace leading to total deadlock, shown in Figure 2 (a final part of the trail leading to deadlock).

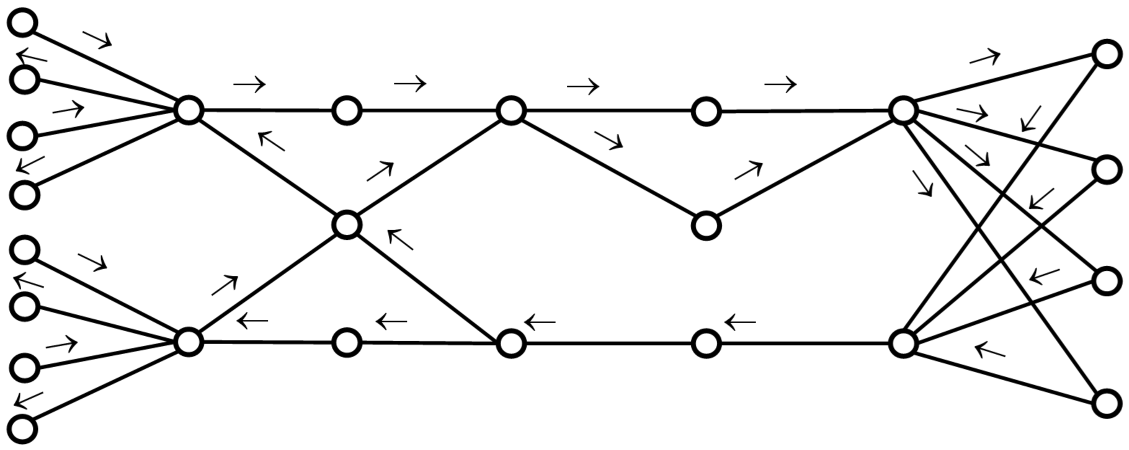

Figure 1. The graph of Rome metro.

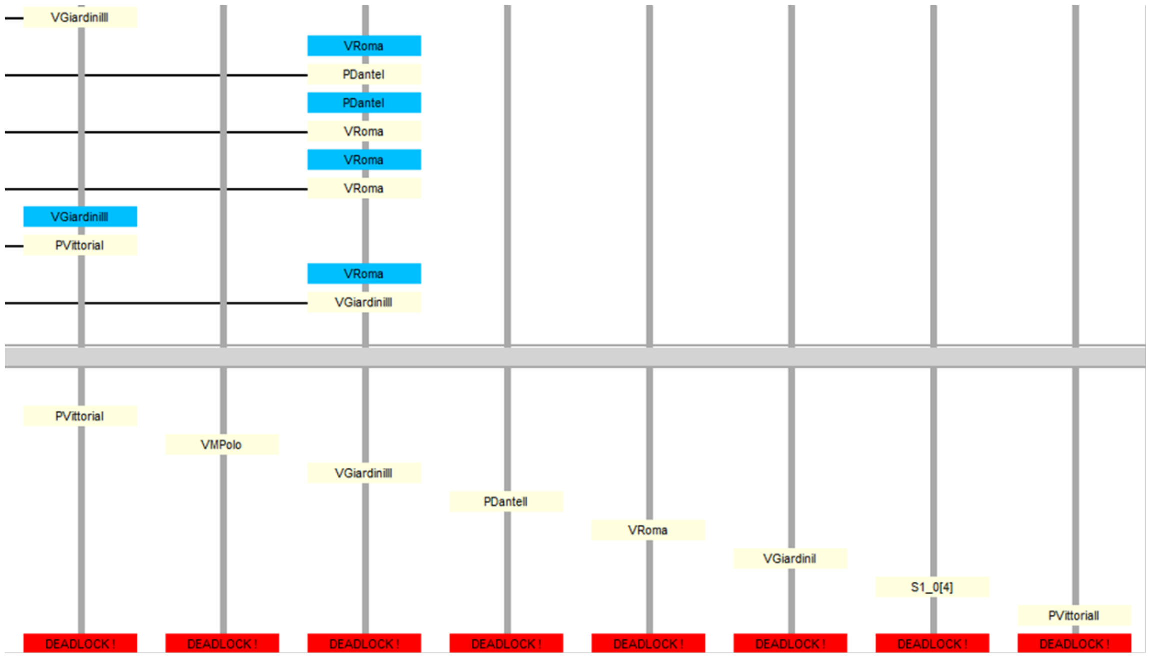

Figure 2. The fragment of the counter-example with a trace leading to a total deadlock of Rome metro. It is the illustration how the counterexample is presented: Vertical lines are timelines of the trains, on which node states and passed messages are shown. Below the thick gray line are the final states of the trains participating in the deadlock and deadlock indicators are presented. The full counter-example is about 30x larger and cannot be fully presented here.

The *DedAn* program constructs the reachability space of the analyzed system, after which it finds possible deadlocks, total or partial, exhaustively searching this space, which is called model checking technique. A combinatorial explosion occurs when determining the reachability space, which limits the possibility of verification to small and medium cases. Many methods of limiting state space have been developed, for example, symbolic model checking, bounded model checking, compositional model checking, local model checking, partial order reduction, abstraction, and slicing [48].

However, very large state spaces cannot be constructed, even using the above methods because of operating memory constraints and computational complexity, explained further in this Section. Non-exhaustive search methods have been developed for such cases, such as genetic algorithms (GA [49]), A* [3] m and ant colony optimization (ACO [3]). Sometimes the methods are supported by machine learning [50]. They allow for detecting total deadlocks, in which all processes are stuck, as a state from which there is no way out. However, they are helpless against partial deadlocks that can occur especially in distributed systems. For the testing of partial deadlocks, in which a subset of processes participates, the algorithm 2-vagabonds was developed which uses two types of search processes. First type vagabond looks for configurations in which a given set of processes is inactive (it possibly falls into a partial or total deadlock, thus we call it a faller). Then it puts the hypothesis that the system is partially deadlocked, and spawns the second type of vagabond. The latter tries to find a configuration in which at least one process from the set escapes out of the inactive state, which falsifies the partial deadlock hypothesis (it tries to "crawl out" from potential partial deadlock, so we call it a crawler). If none of the processes from the set can become active, then the decision is made on the occurrence of a partial deadlock. This decision may, of course, be wrong if the condition falsifying the hypothesis potentially exists (it would be found through an exhaustive search), but it could not be found by a crawler.

So finding a total deadlock (also in other non-exhaustive methods) can be a false negative, and our partial deadlock algorithm can give both false positive and false negative results. However, in practical cases, the procedure is sufficient to find partial deadlocks, and even finding a false deadlock (which has not happened to us) would not disqualify the method, because we have not found in literature a better way to find partial deadlocks in systems for which a reachability space cannot be built for verification.

The presented heuristics can be successfully applied in the distributed computing environment, because fallers and crawlers are almost completely independent of each other, except for the starting point of the latter, and regarding passing the search result. The above example of the Rome metro verification shows a total deadlock when applied to travelling processes view of the system. However, the deadlock is partial if it is observed from the point of view of distributed nodes which represent stations. In such a situation, stations in which the trains are stuck participate in a partial deadlock, while other stations are safe from the deadlock and are ready to accept potential trains.

The use of distributed processing allows for more effective management of spawning vagabonds: Instead of one crawler, which is to verify the partial deadlock hypothesis, a gang of them can be started immediately, which allows faster verification of the hypothesis. In addition, if a faller often finds potential deadlock conditions, it can trigger a gang of its siblings who will help putting hypotheses more effectively.

On the other hand, starting a crawler for every hypothesis may cause a large overhead for running many short distributed processes. The better solution is to start a crawler and to send consecutive deadlock hypotheses to it, served by the crawler one by one. This requires communication between the faller and the crawler not only in the starting and completion of the crawler. Moreover, if the input buffer of the crawler becomes large, it can start its siblings to serve a large number of hypotheses. If any one of the crawlers confirms a partial deadlock, or if the faller finds a total deadlock, which contains partial deadlock of every subset of processes, then all the working crawlers should be killed. Those dependencies give complicated starting, communication and completion structure which is served by our *Shapp* library.

The distributed *DedAn* program now works, and it is still under development. The complicated process structure is a good example of challenges for the *Shapp* library, but it is not a good benchmark for showing the library at work. Moreover, the spread of efficiency of 2-vagabonds algorithm for multiple runs is large because the pseudo-random starting of crawlers, even with repeated seed, gives random results due to the unpredictable relative execution times of crawlers resulting from really random delays, in turn resulting from work in a distributed environment. We discuss the *DedAn* case to show the origins of the mechanisms used in Shapp. However, for the illustration we chose a more regular and well-known example of the ant colony optimization algorithm.

In the basic, non-distributed approach, spawning is done via a mechanism similar to *fork()* function known from POSIX class systems. It is way better than exhaustive search in explicit reachability space approach in terms of model size possible to analyze. For the non-exhaustive algorithm, the problem of 12 dining philosophers can be analyzed in 3 h compared to the exhaustive search algorithm, where the experiment didn't even started due to memory overrun. A maximum case of five philosophers was verified exhaustively.

Soon, it was discovered that the non-exhaustive algorithm allows for promising parallelism, applicable even for a distributed environment. However, plain distributed workload management systems are not designed to solve such problems. Fortunately, the already available *Shapp* library has been providing almost all required functionalities to use DedAn in a distributed environment.

5. The Tool-Shapp Library

The functional requirements for the *Shapp* library were formulated before its creation. Note, that the intention was initially to solve challenges related to *Feniks* simulator. Those are like the following:

1. A program with an attached *Shapp* library should be able to differentiate if the current execution context is related to base (root) or descendant (child) computational unit.

2. The result (as newly created files) of a descendant computational unit should be sent back to its parent unit.
    - The library should support the creation of many descendant units using different approaches to unit creation: Via chaining consecutive programs (different than starting application with *Shapp* library), pool calculations or recursive invocations (in tree-like approach). The nested calls of submitted descendant units might be aware of the expected operating system, required operational memory, needs for processor cores and other software and hardware requirements.
    - The library should inform the user about changes of the descendant unit status asynchronously. This mechanism should cover start, end, suspension and deletion broadcasting to the user of the library.
3. The library should provide an interface for communication between parent and descendant unit. The use case for such mechanism is e.g., telemetry or neural network weights synchronization.
4. The library should return a handle to the newly created descendant unit, e.g., as a descriptor, which will be treated as a differentiator of the computational units among the library. Besides any additional functionalities for computation control and communication, it should contain the numerical ID of the unit.
5. When the application with *Shapp* library is about to close, it should not require the removal of its descendant units. The intention is to allow further processing in a detached state, without a supervisor. However, if the removal of descendant units during application closing is desired, it must be implemented by the user, calling the *kill* operation on the descriptors of descendant units.
6. The library should provide an ability to perform a submission of a simple batch application (or a script) without the *Shapp* library in it.
7. The library should correctly handle shared file systems. In the case of its usage, no unnecessary file transfers should be performed to and from a descendant unit (executables, input, and output files, shared libraries files). Additionally, it case of detected issues with availability in the shared file system, a fallback mechanism to standard file transfer should be used.
8. The library should correctly handle breakdowns of the executors in such a way that a task is restarted as soon as the computational resources are available again. The restarted task instance can be performed on a different executor than during the previous attempt.
9. The library should be able to handle exceptional situations caused by the code written by the user of the library. If an error caused by the software outside the library leads to program termination, the reason of the fail should be recorded as exception stack trace and all the logs and intermediate files should be transferred back to the parent.

5.1. The Operation of HTCondor

Figure 3 shows the state diagram of the HTCondor system for any computational unit processing. The diagram has been prepared by the authors based on the observations of the HTCondor in operation, as well as extracts of information from the documentation [51] and research publications [9].

When execution of the submitted task is finished, the HTCondor system compares the directory tree after the program completion with the starting state. All the recently created files are considered as result files and are considered to be sent back to the node with the parent unit. This behavior can be modified using *Shapp* library, so that only a files specified by the user would be considered to transfer back to the submitter; remaining files will be lost. All the configuration altogether provides a base for the operation of the *Shapp* library—the executable file, input datamx and configuration files are simply transferred to the target node as a set of files in specified locations.

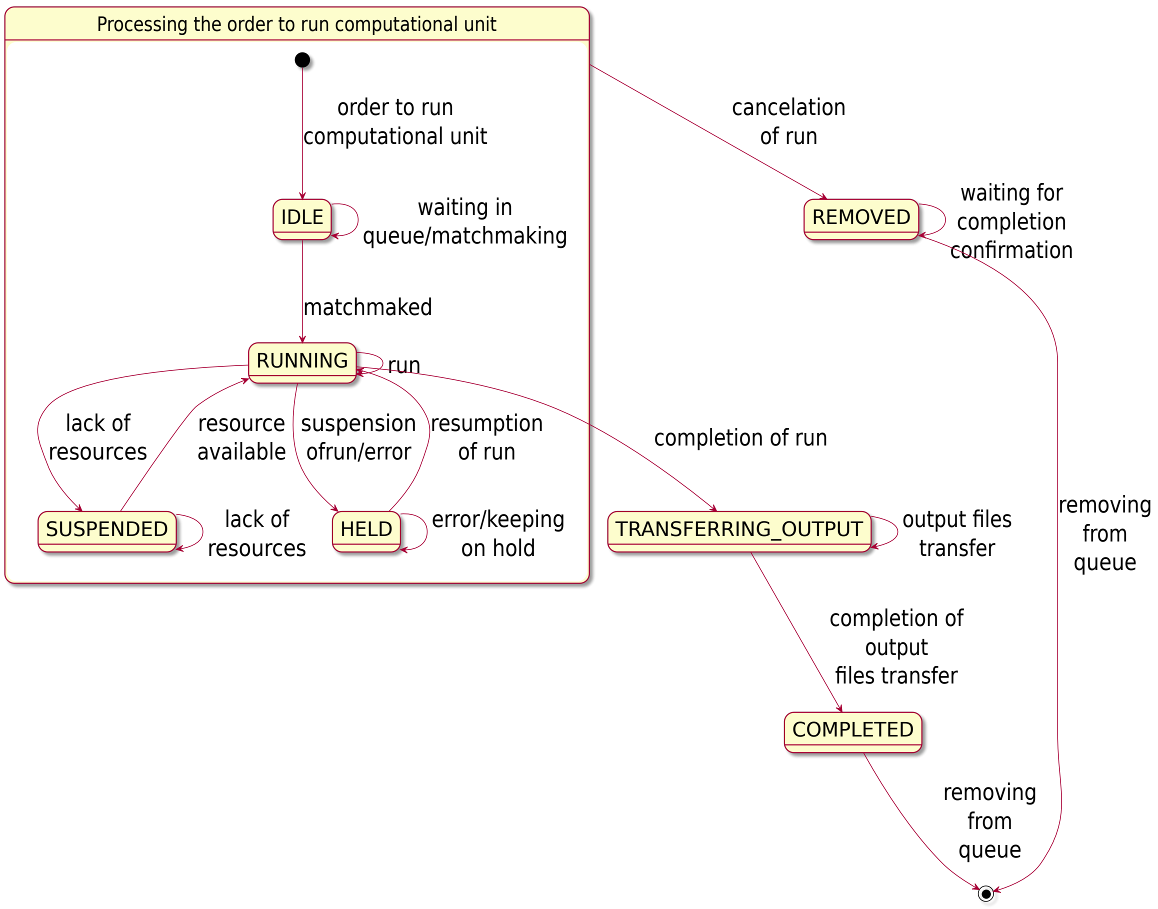

Figure 3. State diagram of tasks in the HTCondor system. The naming convention of the states is exactly the same as the one in HTCondor system documentation.

5.2. Library Adaptation to a New Use Case

As stated in Section 3.1, *Shapp* library was initially designed to work with *Feniks* simulator, where the problem was How to manage the pool of simulations in a comfortable, effective way. However, for *DedAn* program the needs are different: How to perform tree-like processing of a single model in a distributed environment.

Fortunately, the *Shapp* library was designed in a very general way that allows tree-like processing from scratch. For *Feniks* simulator this functionality was not fully utilized. In the simulator, the experiment generator is established which is parameterized with appointing Boolean variables, ranges of integral variables or ranges and steps of real-valued variables. The experiment generator prepares the data structure of the global model and the Cartesian product of values of appointed variables as individual files for simulations. In the original version, Feniksrun experiments using .NET threads. The present, distributed version runs the experiments and controls their completion and results using the *Shapp* library.

In the *DedAn* verifier, the threads run other threads, thus the application of tree-like computing structure is mandatory to be able to use distributed workload management system for this type of problems.

The requirements described in Section 5 were met and sufficient for a massive simulation scenario of Feniks. However, for *DedAn* it is not enough. The list has to be extended with the following properties:

1. The library has to allow two types of termination of the computational unit:

   (a) Soft—when the computational unit has some time (described in the message) to end the computations and clean up the environment.
   (b) Hard—when the computational unit has to terminate without any delay, when the results are out interest or when the environment cleanup has to be done as quickly as possible.

2. A computational unit has to be able to control the execution of its descendant computational units in a hierarchical way. The library should support the following operations without any additional user activities:
   (a) Cut-off branch—terminate execution of all the descendant units and indirectly descendant units of those up to the leaves; notably, with this ability and the one described under Section 4 , a root computational unit can terminate the whole computing tree.
   (b) Detach yourself—after the procedure described in Section 2, terminate the execution of the computational unit which received the order.
3. Descendant computational units have to be able to establish communication with their parent.
4. Every computational unit has to be able to establish communication with the root computational unit.
5. The library has to allow transmissions with arbitrary data structures on the allowed channels between computational units.

With those additional functionalities available, the *Shapp* library can be also used for far more complicated computation flows.

6. Tree-Like Distributed Computations

The main functionality provided by *Shapp* library is tree-like distributed computations. An example of such can be found in Figure 4.

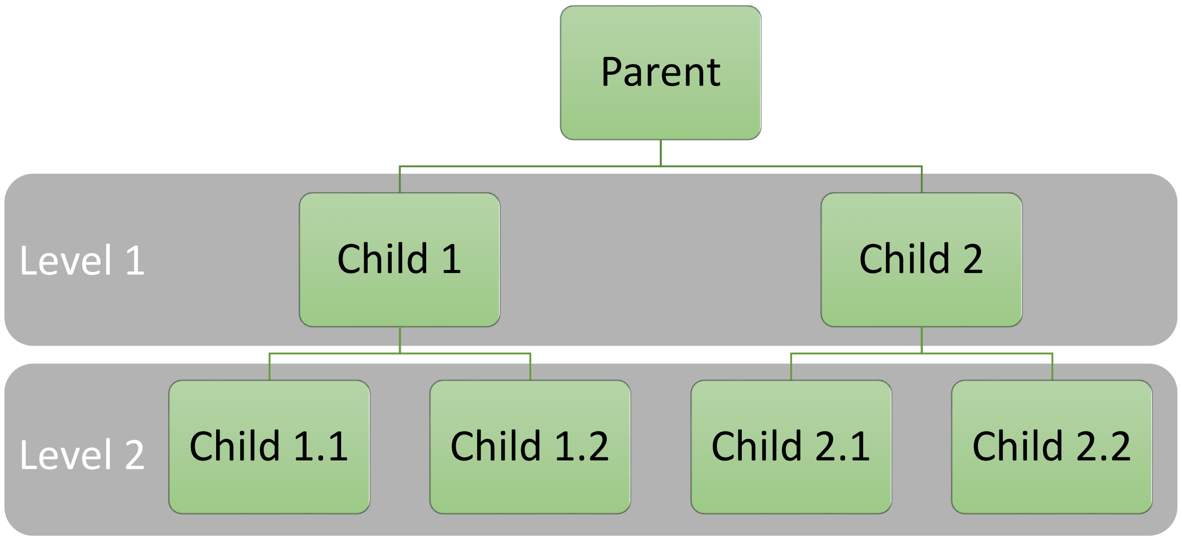

Figure 4. Example of tree-like computation flow. A parent computational unit submits two copies of itself. Then, each child of the parent submits two copies of themselves. When the tasks of level 2 finish their execution, the task of level 1 can also finish.

This solution is similar to the *fork()* function known in POSIX class operating systems. However, there are a few differences:

- The descendant copy starts from the beginning of *main* function, rather than from the branching point.
- The whole program context (stack, static data, code segment, heap) is recreated from scratch, directly from executable file as a new program, instead of reusing it as it happens for *fork()* function, where:

  - Stack (function calls) is copied almost completely [52].
  - Static data (global variables) and heap are copied completely.
  - Code segment (processor instructions which the program consists of) is shared—as it is read-only, no copy is necessary.
- Opened file descriptors are not shared with the descendant copy.

Those limitations are obvious when the implementation is concerned. Underneath the selfSubmitter.Submit(), covered in Listing 1, the *Shapp* library performs a submission into the HTCondor system with a copy of the executable with all the dependencies. It will result in a new, completely independent instance of the program—possibly on a different physical machine. A few example actions possible to do around POSIX *fork()* function with its equivalents using the *Shapp* library were presented in Table 2.

Table 2. Examples of POSIX function calls related to *fork()* function with their equivalents using the *Shapp* library. Note that the descriptor in *Shapp* is the way the library represents unique tasks.

POSIX	**Shapp**

int pid = fork();	var selfSubmitter = new SelfSubmitter(inputFiles, arguments);
	var descriptor = selfSubmitter.Submit();

if (pid == 0) { act as child }
else { act as parent }
	int Main() {
	if (SelfSubmitter.AmIChildProcess()) { act as child }
	else { act as parent }
	}

int pid = wait(NULL);	var descriptor = Shapp.Helper.WaitForAnyJobToEnd(jobDescriptors);
kill(pid, SIGKILL);	descriptor.HardRemove();
exit(0);	Environment.Exit(0);

Thanks to the level of abstraction provided by the library, a copy of the program can be made in just two lines of code. Minimal working example is provided in Listing 1.

Listing 1: Example usage of the *Shapp* library. The root computational unit (root process) submits twice the execution of its copy. Then, all the previously created copies create their own ones again, twice. This results in a hierarchy presented in Figure 4.

```
public static void Main() {
if (AmIRootProcess()) {
SubmitNewCopyOfMyself();
SubmitNewCopyOfMyself();
} else if (GetMyNestLevel() == 1) {
Log("1st␣nest␣level");
SubmitNewCopyOfMyself();
SubmitNewCopyOfMyself();
} else if (GetMyNestLevel() == 2) {
Log("2nd␣nest␣level");
}
}
static void SubmitNewCopyOfMyself() {
var selfSubmitter =
new Shapp.SelfSubmitter();
var descriptor =
selfSubmitter.Submit();
}
```

Thanks to this functionality, it is possible to perform the computations in a tree-like structure despite the fact that in the HTCondor system each of the computational units has equal priority and does not have any hierarchy provided by the system. The order of processing is not strictly determined in HTCondor.

6.1. Communication

HTCondor is a distributed workload management system of high-throughput class. It has been designed to run as many power-consuming tasks as possible, also called batches. The main assumption here is that those tasks are completely independent from each other. However, thanks to the *Shapp* library this limitation can be overpassed.

6.1.1. Static Interface

Basic level of communication is provided via mechanism available already in HTCondor. When the computational unit is being submitted, additional files (created and pointed by the user of the *Shapp* library) can be also provided in a package with the executable. In the case of a *DedAn* non-exhaustive search, when a crawler process (described in Section 4) is submitted, the input files are:

- A model the verified system expressed in *DedAn* input language.
- A path leading from the initial configuration to the starting configuration of crawler activity.

In addition to the mentioned files, the *Shapp* library is also able to provide additional program call arguments (known as argv). Those are used to pass the file names provided with the executable.

The files are also used for communication in the opposite direction. When any computational unit executed via HTCondor ends its execution, the system transfers automatically all the newly created files back to the submitter—in the case of Shapp, it is the parent computational unit.

This simple method is used currently to pass the results of processing of a crawler vagabond (described in Section 4) back to the faller vagabond. The results are:

- A file with the return code of the agent which determines the result of verification of the deadlock hypothesis (true or false).
- A file containing a counter-example, if the return code informs that that the deadlock is found.

Based on the result of the completed crawler vagabond, the faller can decide about killing all the remaining computational units currently being processed.

6.1.2. Dynamic Interface-Intranet Communication

In the future there will be also more advanced interface available. This communication structure will be based on intranet configuration and will provide dynamic communication. It will be used for more advanced mechanisms, closer described in Section 8. Available channels are shown in Figure 5.

Intranet communication will be realized using regular TCP sockets with client-server architecture.

- A server is hosted always by a computational unit. It is started at the beginning of the main function, before submission of any descendant unit. There is only one instance of a server started and its responsibility is to handle all the descendant and sibling units at the same time.
- A client is hosted always by descendant computational unit. It always connects to a server in its parent unit and in the root unit. After establishing the connection to the parent, if the descendant computational unit has any siblings, it also establishes connection to all its siblings. The addresses of the siblings are obtained from the parent unit.

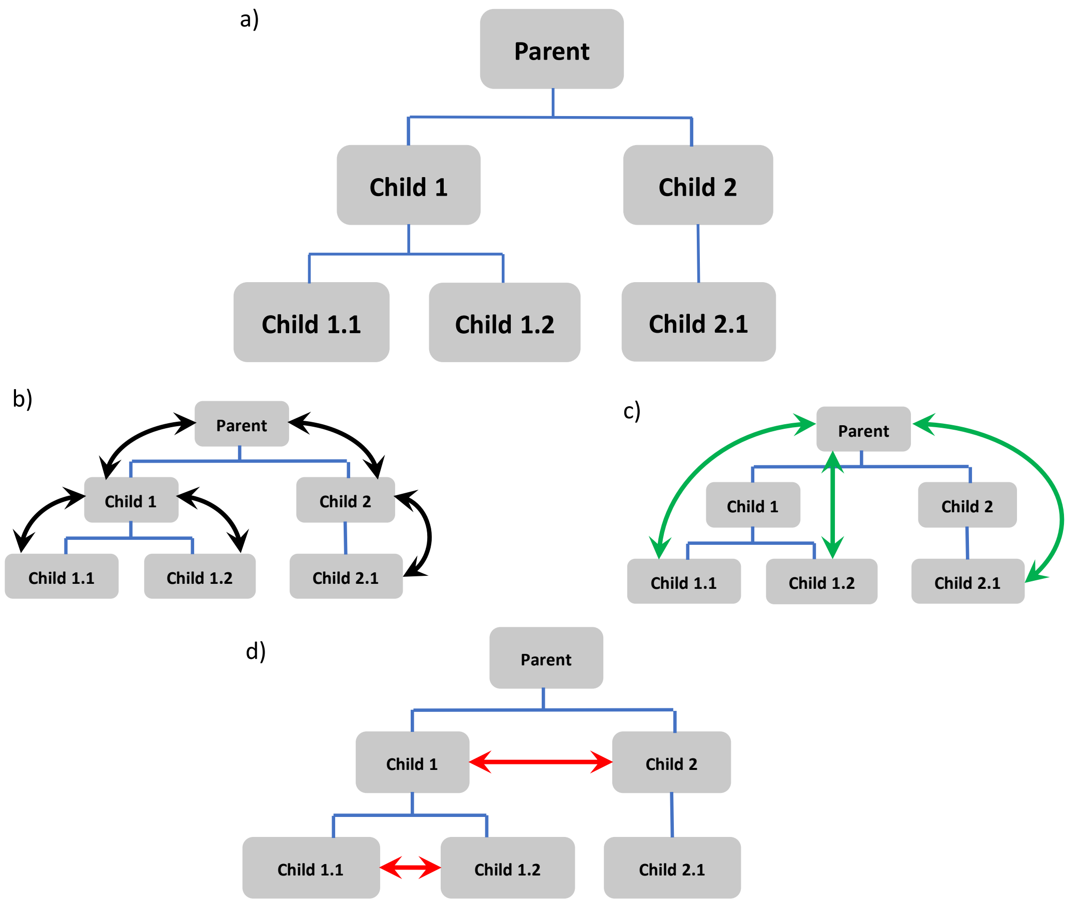

Figure 5. The diagram of possible channels between the computational units. Figure **(a)** example computation tree structure; **(b)** the basic way of communication, between parent and its children; **(c)** any-to-root channels. **(d)** channels established between siblings of the same parent. The channels are realized using TCP sockets. The IP address and port ID are passed as environmental variables for each created child.

The described communication channels have to be established during the child computational unit startup. However, TCP sockets allow for sending arbitrary, binary data. To ensure easy protocol scaling and ease of the library usage, the following wrappers were considered for protocol specification base:

1. Google Protobuf [53].
2. C# built-in XmlSerializer [54].
3. C# built-in BinarySerializer [55].

Just three of the most popular representatives of major serialization solutions were considered. The properties of each serialization mechanism are described in Table 3.

Table 3. List of considered protocol wrappers with its main properties.

	Google Protobuf	**XmlFormatter**	**BinaryFormatter**
Description language	Dedicated language	C#	C#
Message representation	Binary	Text	Binary
Size efficiency	Very high	Low	High
Backward compability	Yes	No	No
Polymorphism	No	Yes	Yes
Supported languages	i.a. C#, C++, Java	C# only	C# only

In terms of universality, effectiveness, and popularity, Google Protobuf is definitely a winner. On the other hand, XmlFormatter is quite easy to use natively in C#, and produced messages are human-readable, which is considerably helpful in debugging and observation. However, resulting message sizes are significantly higher, and may result in throughput issues in a high-scale distributed system. BinaryFormatter is a mid-step between effectiveness of Protobuf and ease of use of XmlFormatter. It is also easy to use natively in Shapp—the C# library and relatively efficient compared to Protobuf [56]. Nonetheless, BinaryFormatter is not a perfect solution and has some drawbacks:

- Lack of support for other programming languages prevents the integration of *Shapp* library into other programming languages, preserving its compatibility with the C# version.
- Lack of backward compatibility disallow communication between different versions of the library; however, given that the main application of *Shapp* is cloning the entire program, always with the latest libraries—this is not a real problem.
- Lack of human-readable messages can make debugging of the distributed application harder; still, this can be done using the C# debugger—as all the messages are just language class instances. Debugging communication between instances of the distributed program remains a problem, we do this by tracing the protocol steps to log files.

Regardless of the minor drawbacks which can be overwhelmed, BinaryFormatter offers an important facility—support for polymorphism. This will help with the *Shapp* library protocol extension and design process.

In the traditional approach of protocol handling, the messages are processed in a switch, which behaves as a matcher of the action for a particular type of protocol message [57]. In this approach when a protocol is modified by introducing a new message, the change has to be done in two fragments:

- In the file with structures definition (an example in Listing 2).
- In the main place with reaction definitions on the newly created message type (an example in Listing 3).

Listing 2: An example of first necessary modification in the switch approach to protocol definition. Here, in some separate file with structures definition, a new structure has to be added with a new message in a protocol.

```
struct HelloFromChild {
JobId MyJobId;
}
```

Listing 3: An example of necessary modifications in the switch approach to protocol definition can be seen here. In the file with reaction definitions it is necessary to modify 3 code fragments: Enumeration with message differentiators, main union with all the possible message types to be received and the main dispatch function, where a new case statement has to be added for a newly created structure. Note that this is not a valid C# code, unions have to be defined in a different way. To avoid obfuscation, the example was simplified to a C-style union.

```
enum MsgNo {
...,
HelloFromChildNo
}
union Message {
MsgNo no;
...
HelloFromChild helloFromChild;
}
```

```
void dispatch(Message msg) {
switch (msg.no) {
...
case HelloFromChildNo:
dispatch(msg.helloFromChild);
break;
}
}
void dispatch(HelloFromChild helloFromChild) {
// the reaction on newly added message
}
```

Additionally, such dispersion of logic and data may lead to inconsistencies. As for a single logical change (e.g., new message type addition), it is necessary to modify at least two files in four locations (as shown in Listings 2 and 3). Opposition to this approach is a polymorphic solution, when the message data and reaction of the system on this message reside in the same code fragment. This mechanism, called polymorphic dispatch, was used in *Shapp* library for the whole internal protocol specification.

6.1.3. Polymorphic Dispatch

The *Shapp* library treats parts of the protocol message types as independent entities, realized as classes implementing a common *ISystemMessage* interface. This interface is presented in Listing 4.

Listing 4: The main interface used as a base for all the protocol messages used by *Shapp* library for dynamic mechanisms like cut-off branch and soft termination described in Section 5.2. The only method, *Dispatch* with sender as the argument, is required to be defined in the protocol classes. This method describes the behavior on the reception of the instance of the class. It will be called right after message reception.

```
interface ISystemMessage {
void Dispatch(Socket sender);
}
```

On top the main interface, all the remaining parts of the interface are created. Example protocol message definition is presented in Listing 5.

Listing 5: Example *Shapp* protocol message implementation. In this example, there is a simple ping response. The behavior on *HelloFromChild* message reception is described in its *Dispatch* method. Additionally, the message can have an arbitrary number of additional fields that will carry the data (here, *MyJobId* field) and common for all instances, static fields. Here, the static field *OnReceive* is used for an asynchronous notification of a new message of this class being received.

```
[Serializable]
class HelloFromChild : ISystemMessage {
static event Callback OnReceive;
JobId MyJobId;
void Dispatch(Socket sender) {
OnReceive?.Invoke(sender, this);
}
}
```

After such preparation of the interface, the polymorphic dispatch can be applied. It is done in a standard way as for any other example of polymorphism, as presented in Listing 6.

Listing 6: The polymorphic dispatch in a real application. Opposition to the switch approach to protocol behaviour description, here no changes are needed in the dispatcher code. The only requirement is that newly added message types have to implement the interface *ISystemMessage*, presented in Listing 4.

```
if (AmIRootProcess()) {
var server = new AsynchronousServer();
var dispatcher = (receivedObject, socket) => {
if (receivedObject is ISystemMessage sysMsg)
sysMsg.Dispatch(socket);
};
server.NewMessageReceivedEvent += dispatcher;
server.Start();
}
```

Thanks to such an approach to protocol definition, important goals are achieved:

- The protocol behavior is encapsulated to particular message definitions.
- The protocol can be easily extended, without affecting already present message types.
- Definitions of the behavior and the data are close to each other in terms of location in the source code (single class space).
- Complex data types built-in into the C# language can be used without additional effort in the protocol messages, e.g., Dictionaries, HashTables, Sets, Matrices, etc.
- Already existing classes can easily gain the ability to become a part of the protocol; an example is a class defining a huge set of parameters for the computational program, such as a simulator. To be able to transfer this class over *Shapp* library protocol, the class has to simply extend the ISystemMessage interface.

All the described properties have gained *Shapp* library great opportunities for further development and abilities extension when new needs and expectations arrive.

7. Experiment

For the experiment with tree-like computing and communication via polymorphic dispatch (described in Section 6.1.3), the traveling salesman problem [58] was used (finding the cheapest full cycle in an undirected graph with costs assigned to the edges). The problem was solved using the ant colony algorithm [3]. The algorithm is used to find the optimal path in a graph. Paths in the graph are called ant routes. The segments of these paths are marked with certain weights called pheromones. Pheromones evaporate after some time. Other ants prefer segments with the pheromone laid down. The decision to use ACO was taken based on the following conclusions:

- To be able to make a comparison, the problem should be easy to realize in three environments:
  - Single machine, using:
    - Threads.
    - The *Shapp* library.
  - Multiple computers, using the *Shapp* library.
- Tasks realized on all the workers should be similar to avoid unnecessary noise in the results.
- Tasks should produce repeatable results.
- tasks should require some way of communication and synchronization between workers about their state and current parameters; however, the synchronization process should be periodic, with a frequency lower than 1 Hz to avoid overhead on workers during messages serialization (as the messages can be arbitrarily large, even for a small graph of 500 cities each message was over 2 MB).

The main aim of this experiment was to verify whether distributed computing using *Shapp* library in a tree-like manner had the ability or potential to speed up model solving process for *DedAn* program in the future. Additionally, the influence of the overhead of HTCondor [11] system (the base of the *Shapp* library) was taken under analysis.

7.1. Environment

The experiment was done in three hardware configurations:

- For *threading—single server* and for *Shapp—single server*: A single server with the following parameters:
  - OS—Microsoft Windows Server 2008 (as a virtual machine).
  - CPU—Intel Xeon CPU E5-2630@2.30 GHz, Cores: 6, Threads: 12.
  - RAM—3.34 GB of virtualized memory.
- For *Shapp—separate computers* workers run in a set of two computers with the following parameters:
  - OS—Microsoft Windows 10.
  - CPU—Intel Core i5-4670 CPU@3.40 GHz, Cores: 4, Threads: 4.
  - RAM—32 GB DDR3.
- For *Shapp—lab* workers run on a set of total 18 computers in a single lab in the following configuration:
  - A set of six computers with the following parameters:
    - OS—Microsoft Windows 10.
    - CPU—Intel Core i5-4670 CPU@3.40 GHz, Cores: 4, Threads: 4.
    - RAM—32 GB DDR3.
  - A set of 12 computers with the following parameters:
    - OS—Linux CentOS 7.
    - CPU—Intel Core i5-8400 CPU @ 2.8GHz, Cores: 6, Threads: 6.
    - RAM—32 GB DDR4.
    - .NET execution environment—mono [59].

The first two hardware configurations were used in order to accomplish the following goals:

- A single server instance is used to compare the same task done using system threads and *Shapp* library. The main purpose of it is to analyze the influence of distributed workload management system usage on time and performance overhead, compared to threading solution.
- A set of individual computers is used to analyze the influence of distribution of the task on a set of weaker computers and the influence of network transmissions over LAN with the task parameters.

The third configuration was used in order to present computation scaling ability using *Shapp* library and HTCondor.

All the computations made with *Shapp* library were using HTCondor as the executor of the tasks. It is important to note, that extending the pool of available slots (in HTCondor nomenclature: A pair of one CPU thread and $\frac{1}{N}$ of operating memory, where N describes the number of CPU threads) is as easy as installation of HTCondorsystem on another server and appropriate configuration.

7.2. Conducting the Experiment

The aim of the computations was to find the shortest trail over all the cities in the model. The graph is modeled in a random way; however, the random generator seed is always the same. The realized task has the following properties:

- The graph represents 500 cities.
- There are three workers realizing ACO algorithm.
- Each worker is holding and processing five ants,.
- The pheromone values on the edges are synchronized between workers once per 10 s.
- The algorithm runs for 240 s.

The computations were performed in two ways—via using threads, and via the *Shapp* library (on a single computer and in lab network). The implementation details can be found on the *Shapp* project repository [60], in the example project shipped with the library itself.

7.3. Results

The first parameter being captured during the test was the number of successfully performed iterations of ACO algorithm. Each iteration consists of building up new trails for each ant and pheromone update on graph edges. However, *Shapp* workers have to perform best trail checking once per second, whereas in the threading case this task is delegated to a different core, also doing the same thing once per second. This leads to a longer average time spent on a single iteration. Additionally, each *Shapp* worker has to have an instance of the TCP client socket running, as well as once per 10 s it has to send its status with the best found trail and its changes in pheromone on graph edges.

All of the mentioned differences lead to a processing time in *Shapp—single server* that is higher by around 10% than in the *threading—single server* scenario, as it can be seen in Figure 6. However, note that such difference is caused by the lack of shared memory, which greatly reduces the amount of communication necessary in the threading scenario to keep the model coherent. The difference between both *Shapp* scenario is caused by the hardware differences. Even though the *Shapp—single server* used a better CPU, the virtualization overhead led to performance better about 1.8 times in the *Shapp—separate computers* case. On the other hand, the *Shapp—lab* scenario was able to perform around 23 times more iterations than the *Shapp—separate computers* case, as it can be seen in Figure 7.

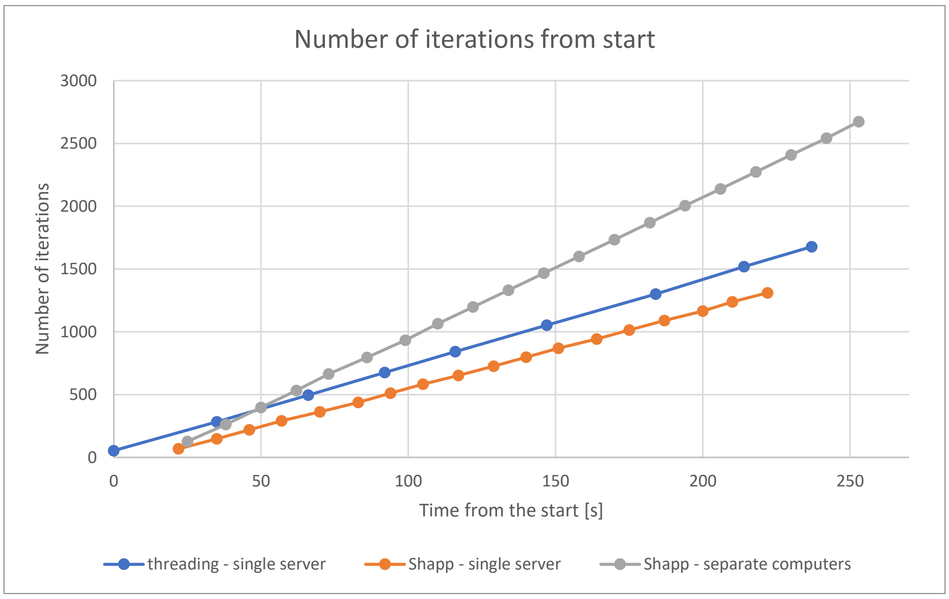

Figure 6. The plot of the number of updates done to the ant colony optimization (ACO) model over time. *Shapp—single server* plot is shifted to the right by 22 s and *Shapp—separate computers* by 25 s, which stands for the delay introduced by the HTCondor system.

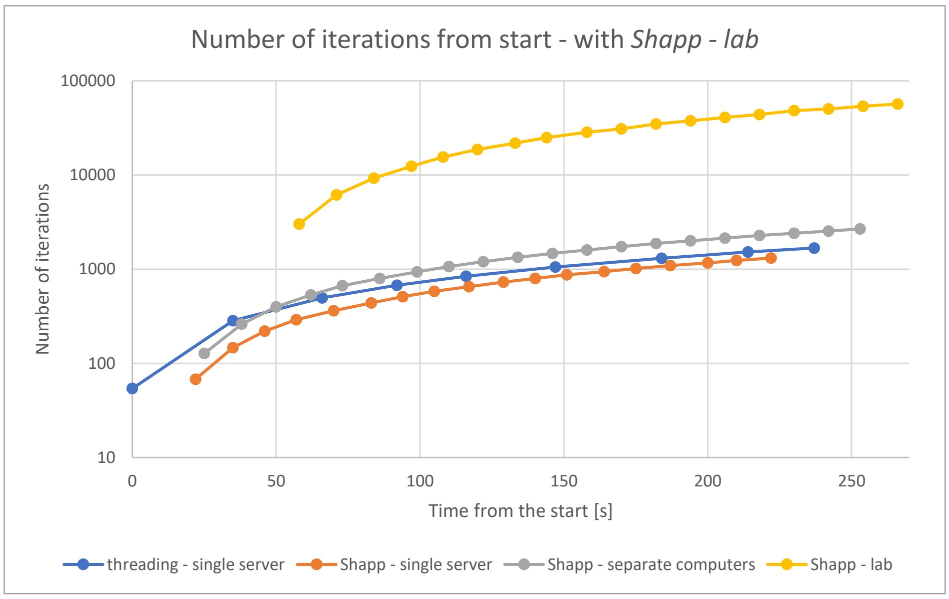

Figure 7. Extension of the plot in Figure 6 with data series for iterations from start for *Shapp—lab*, which reached the value of 56,674 successful iterations. It is shifted to the right by 58 s which stands for the delay introduced by the HTCondor system. Ordinate axis scale was changed to logarithmic to improve the readability of the plot.

Also, it has to be expressed that the *Shapp—single server* number of iterations plot is intentionally shifted 22 s to the right. This was exactly the time that was necessary to setup all the workers using *Shapp* library to be ready to start the computation. On the other hand, in the *Shapp—separate computers* scenario, this delay took 25 s and in the *Shapp—lab* scenario, it took 58 s.

The second analyzed parameter was the behavior of the best trail discovered in the last second of the execution. The plot of it is presented in Figure 8. The first important issue visible here is that the *Shapp—single server* scenario has reached the best trail length at the 94 s time mark, whereas the *threading—single server* scenario has reached a similar value almost at the end of the experiment, at the 237 s time mark; despite the fact that the *Shapp—single server* scenario required 22 s more for workers startup. The *Shapp—separate computers* has reached the best trail length (501) in the last measurement, at the 253 s time mark. However, since the 170 s mark, the best trail length reached 509, and after this moment improvements became noticeably slower. On the other hand, *Shapp—lab* reached the best trail length (524) at the 84 s time mark and didn't improve more till the end of the experiment.

However, the shape of the plot lines varies. The threading line approves the optimal value in a much smoother manner, reaching value 553; the same applies to *Shapp—separate computers* scenario, where value 501 was reached. Whereas in *Shapp* case, the best trail is reached much faster and stabilizes soon at value 554. The reason for such behavior is the conjunction of the following components:

- ACO algorithm parameters were not tuned very well (as this was not the aim of this experiment).
- Even though the synchronization of the pheromone along the workers was in all the scenarios done each 10 s, in both *Shapp* scenarios there were present significant delays (due to TCP transmissions).
- Comparing the *Shapp* scenarios, separate computers had more computational capacity, as can be seen in Figure 6.

Additionally, the *Shapp—lab* scenario approached the optimal value in a similar manner to the *Shapp—single server* case. The reason for such behavior can be found in ACO parameters, which were the same for each scenario. Most probably, for the *Shapp—single server* the pheromone amplification parameter was not big enough, whereas for the *Shapp—lab* it was too large considering the number of workers.

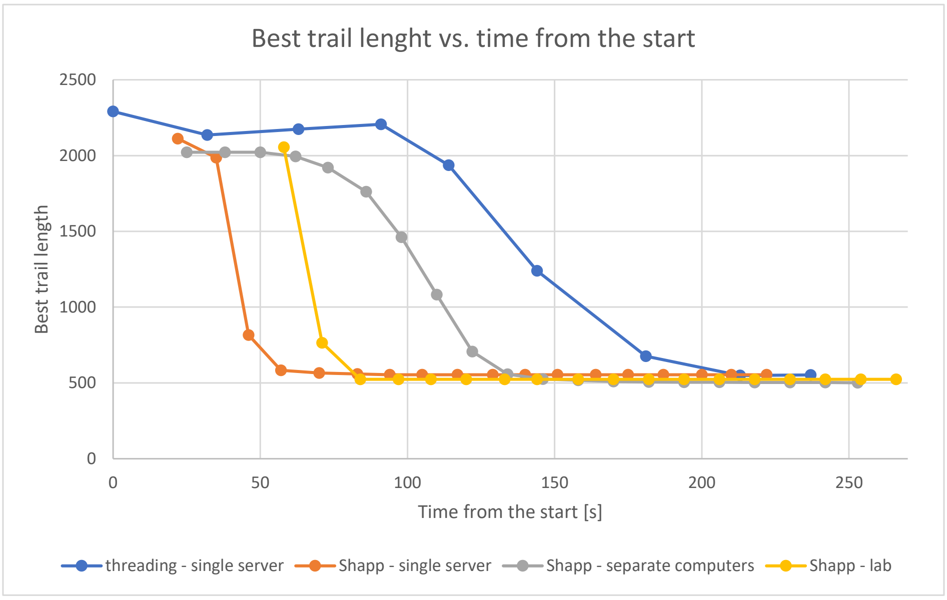

Figure 8. The plot of best trail of all available from time. *Shapp—single server* plot is shifted to the right by 22 s and *Shapp—separate computers* by 25 s, which stands for the delay introduced by HTCondor system.

7.4. Experiment Conclusions

Even though computations performed using system threads are much more efficient and easier to handle and communicate, they are not capable of further scaling up of the computations (except for hardware updates). In the *Shapp* case, however, usage of HTCondor system allows for easy scaling of the computation pool capacity by adding new hosts to the pool. And thanks to additional functionalities provided by *Shapp* library, collected in Table 2, usage of distributed workload management system in processing can be almost as easy as it is in the threading approach.

8. Conclusions and Further Studies

The article presents the workload management system using the original *Shapp* library, based on the HTCondor. Unlike existing systems of this type, which simply allow for running executable programs, our library makes it possible to run tasks from the programming language level, in this case, C#. It is possible to run tasks in the legacy style, that is, to run programs on available cores in a distributed system, and to control their state and finish calculations. This style of execution suits massive simulations, as in the PRT *Feniks* simulator (Section 5.2), in which it is necessary to carry out hundreds of simulations of the same model for different sets of parameters and to collect results.

However, many computational problems require more accurate control of running tasks, such as finalizing calculations for some tasks when better promising calculation directions are found. Taking the management to the level of the library available in the programming language and adding certain functions to manage child tasks, such as checking the request to finish the task or task with its descendants, the efficiency of running, and checking the completion of tasks, allows for better control of the state of calculations. This provides a task tree or another complex calculation structure, for example, to examine the tree space of solutions with the possibility of pruning. An example is the algorithm of non-exhaustive search of the state space in the algorithm of partial deadlock detection of 2-vagabonds. The basic faller process can run a pack of its brothers as needed, and can run any number of crawlers to verify deadlock hypotheses. Crawlers can be equipped with input buffers to reduce the cost of their cyclical startup, but they can also run their brothers when the input buffers are overloaded. Other, complex calculation structures richer than tree are also possible.

Finally, communication between tasks allows for their effective management, dynamic configuration of their work, change of their input parameters during execution, and in the opposite direction reporting of partial results or reconciliation of decisions with tasks of the same level. All these capabilities are available through operations similar to the fork, wait, exit, and kill functions in the POSIX specification, with restrictions due to distribution. A common feature with the POSIX specification is the ability to run process clones, operating on the same global data structure (in the library passed as a common input file), but with its own logic and work parameters, passed as call parameters and files parameterizing the child's work.

The use of the library in the *Shapp* simulator and *DedAn* verifier showed significant usability in both the parallelization of calculations and the management of massive, parallel calculations. The benchmark in the form of an ant algorithm shows the ease of using the library and managing distributed tasks. Additionally, the experiment proved that a distributed approach to workload management does not introduce significant overhead, instead of keeping it around 10%. However, at the same time, HTCondor allows for easy and cheap scaling of the computational pool. Such scaling can be done using regular computers, which during the day are used in regular day-to-day tasks, and during the night utilized as outright participators in the research-requiring high-throughput computing system.

The current state of the *Shapp* project is only the basis for further research. The ability to control the flow of calculations in a distributed environment opens up completely new possibilities for intelligent resource allocation for calculations. In the further development of the library, communication functions will be developed, as well as the ability to group tasks and run a task group in a single operation, and the group management of tasks. The most promising area of research is the ability to modify heuristics used in algorithms to more efficiently use available resources.

Author Contributions: Writing—original draft preparation, T.G.; writing—introduction, conclusions and *DedAn* description, W.B.D.; writing—review and editing, W.B.D.; supervision, W.B.D.; conception, T.G. and W.B.D.; methodology, T.G. and W.B.D.; experiments, T.G. and W.B.D.; software, T.G.; resources, T.G.; visualization, T.G.; validation, W.B.D.; IMDS formalism and 2-vagabonds algorithm, W.B.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Acknowledgments: The authors thank Mariola Jamiołkowska, Rafał Bajorek, Waldemar Grabski and Zbigniew Szymański for their help in preparation of the experiments.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

Shapp	open source C# library for distributed workload management system control
HTCondor	distributed workload management system
DedAn	deadlock analyzer program
ACO	ant colony optimization
GA	genetic algorithm
POSIX	Portable Operating System Interface for UNIX, family of standards describing an operating system API

References

1. Liu, H.; Wang, K.; Luo, J.; Chen, Z.; Yang, B.; He, R. Accelerating Large-scale Reservoir Simulations Using Supercomputers. In Proceedings of the 78th EAGE Conference and Exhibition, Vienna, Austria, 30 May 2016; pp. 1–5. [CrossRef]
2. Daszczuk, W.B.; Mieścicki, J.; Grabski, W. Distributed algorithm for empty vehicles management in personal rapid transit (PRT) network. *J. Adv. Transp.* **2016**, *50*, 608–629. [CrossRef]
3. Francesca, G.; Santone, A.; Vaglini, G.; Villani, M.L. Ant Colony Optimization for Deadlock Detection in Concurrent Systems. In Proceedings of the IEEE 35th Annual Computer Software and Applications Conference, Munich, Germany, 18–22 July 2011; pp. 108–117. [CrossRef]

4. Daszczuk, W.B. 2-Vagabonds: Non-Exhaustive Verification Algorithm. Available online: https://www.springer.com/gp/book/9783030128340 (accessed on 2 March 2020).
5. Baumann, A.; Appavoo, J.; Krieger, O.; Roscoe, T. A Fork() in the Road. Available online: https://www.microsoft.com/en-us/research/uploads/prod/2019/04/fork-hotos19.pdf (accessed on 2 March 2020).
6. HTCondor Main Page. Available online: https://research.cs.wisc.edu/htcondor/ (accessed on 2 March 2020).
7. Hollowell, C.; Barnett, J.; Caramarcu, C.; Strecker-Kellogg, W.; Wong, A.; Zaytsev, A. Mixing HTC and HPC Workloads with HTCondor and Slurm. *J. Phys. Conf. Ser.* **2017**, *898*, 082014. [CrossRef]
8. Du, R.; Shi, J.; Zou, J.; Jiang, X.; Sun, Z.; Chen, G. A Feasibility Study on workload integration between HT-Condor and Slurm Clusters. *EPJ Web Conf.* **2019**, *214*, 08004. [CrossRef]
9. Fajardo, E.M.; Dost, J.M.; Holzman, B.; Tannenbaum, T.; Letts, J.; Tiradani, A.; Bockelman, B.; Frey, J.; Mason, D. How much higher can HTCondor fly? *J. Phys. Conf. Ser.* **2015**, *664*, 062014. [CrossRef]
10. Erickson, R.A.; Fienen, M.N.; McCalla, S.G.; Weiser, E.L.; Bower, M.L.; Knudson, J.M.; Thain, G. Wrangling distributed computing for high-throughput environmental science: An introduction to HTCondor. *PLoS Comput. Biol.* **2018**, *14*, e1006468. [CrossRef]
11. Fajardo, E.; Würthwein, F.; Jones, R.; Philpott, S.; Strosahl, K. Limits of the HTCondor Transfer System. *EPJ Web Conf.* **2019**, *214*, 03008. [CrossRef]
12. Gałecki, T.; Daszczuk, W.B. Shapp: Workload Management System for Massive Distributed Calculations. In *Software Engineering Methods in Intelligent Algorithms*; Radek, S., Ed.; Springer: Prague, Czech, 2019; pp. 318–329. [CrossRef]
13. Sfiligoi, I. glideinWMS—A generic pilot-based workload management system. *J. Phys. Conf. Ser.* **2008**, *119*, 062044. [CrossRef]
14. Pinthong, W.; Muangruen, P.; Suriyaphol, P.; Mairiang, D. A simple grid implementation with Berkeley Open Infrastructure for Network Computing using BLAST as a model. *PeerJ* **2016**, *4*, e2248. [CrossRef]
15. Cameron, D.; Wu, W.; Bogdanchikov, A.; Bianchi, R. Advances in ATLAS@Home towards a major ATLAS computing resource. *EPJ Web Conf.* **2019**, *214*, 03011. [CrossRef]
16. Ďurech, J.; Hanuš, J.; Vančo, R. Asteroids@home—A BOINC distributed computing project for asteroid shape reconstruction. *Astron. Comput.* **2015**, *13*, 80–84. [CrossRef]
17. Patoli, M.Z.; Gkion, M.; Al-Barakati, A.; Zhang, W.; Newbury, P.; White, M. An open source grid based render farm for blender 3d. In Proceedings of the IEEE/PES Power Systems Conference and Exposition, Seattle, MA, USA, 15–18 March 2009; pp. 1–6. [CrossRef]
18. Chorazyk, P.; Byrski, A.; Pietak, K.; Kisiel-Dorohinicki, M.; Turek, W. Volunteer computing in a scalable lightweight web-based environment. *Comput. Assist. Methods Eng. Sci.* **2017**, *24*, 17–40.
19. Pellegrini, S.; Giacomini, F.; Ghiselli, A. A Practical Approach for a Workflow Management System. In *Grid Middleware and Services: Challenges and Solutions*; Domenico, T., Ramin, Y., Wolfgang, Z., Eds.; Springer: Boston, MA, USA, 2008; pp. 279–287. [CrossRef]
20. Anderson, D.P. BOINC: A System for Public-Resource Computing and Storage. In Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing, Pittsburgh, PA, USA, 8 November 2004; pp. 4–10. [CrossRef]
21. Haussmann, J.; Blochinger, W.; Kuechlin, W. Cost-efficient parallel processing of irregularly structured problems in cloud computing environments. *Clust. Comput.* **2019**, *22*, 887–909. [CrossRef]
22. Raicu, I.; Foster, I.; Zhao, Y.; Szalay, A.; Little, P.; Moretti, C.M.; Chaudhary, A.; Thain, D. Towards data intensive many-task computing. In *Data Intensive Distributed Computing: Challenges and Solutions for Large-Scale Information Management*; Kosar, T., Ed.; IGI Global: Buffalo, NY, USA, 2012; pp. 28–73. [CrossRef]
23. Pool, M. Distcc, a Fast Free Distributed Compiler. 2003. Available online: https://distcc.github.io/ (accessed on 2 March 2020).
24. Zhang, W. Linux Virtual Server for Scalable Network Services. Available online: http://www.linuxvirtualserver.org/ols/lvs.pdf (accessed on 2 March 2020).
25. Martin, A.; Gamess, E.; Urribarri, D.; Gómez, J. A Proposal for A High Availability Architecture for VoIP Telephone Systems based on Open Source Software. *Int. J. Adv. Comput. Sci. Appl.* **2018**, *9*, 1–11. [CrossRef]
26. Han, M.; Yao, D.G.; Yu, X.L. A Solution for Instant Response of Cloud Platform Based on Nginx + Keepalived. In Proceedings of the International Conference on Computer Science, Communications and Multimedia Engineering, Beijing, China, 24–25 March 2019. [CrossRef]

27. Matos, R.; Dantas, J.; Araujo, J.; Trivedi, K.S.; Maciel, P. Redundant eucalyptus private clouds: Availability modeling and sensitivity analysis. *J. Grid Comput.* **2017**, *15*, 1–22. [CrossRef]
28. Sun, N.H.; Bao, Y.G.; Fan, D.R. The rise of high-throughput computing. *Front. Inf. Technol. Electron. Eng.* **2018**, *19*, 1245–1250. [CrossRef]
29. Assiroj, P.; Hananto, A.L.; Fauzi, A.; Hendric Spits Warnars, H.L. High Performance Computing (HPC) Implementation: A Survey. In Proceedings of the Indonesian Association for Pattern Recognition International Conference (INAPR), Tangerang, Indonesia, 7–8 September 2018; pp. 213–217. [CrossRef]
30. Valero-Lara, P.; Nookala, P.; Pelayo, F.L.; Jansson, J.; Dimitropoulos, S.; Raicu, I. Many-task computing on many-core architectures. *Scalable Comput. Pract. Exp.* **2016**, *17*, 32–46. [CrossRef]
31. Brunin, G.; Ricci, F.; Ha, V.A.; Rignanese, G.M.; Hautier, G. Transparent conducting materials discovery using high-throughput computing. *NPJ Comput. Mater.* **2019**, *5*, 1. [CrossRef]
32. Anderson, E.J.; Linderoth, J. High Throughput Computing for Massive Scenario Analysis and Optimization to Minimize Cascading Blackout Risk. *IEEE Trans. Smart Grid* **2017**, *8*, 1427–1435. [CrossRef]
33. Bird, I. Computing for the Large Hadron Collider. *Annu. Rev. Nucl. Part. Sci.* **2011**, *61*, 99–118. [CrossRef]
34. Correa-Baena, J.P.; Hippalgaonkar, K.; van Duren, J.; Jaffer, S.; Chandrasekhar, V.R.; Stevanovic, V.; Wadia, C.; Guha, S.; Buonassisi, T. Accelerating Materials Development via Automation, Machine Learning, and High-Performance Computing. *Joule* **2018**, *2*, 1410–1420. [CrossRef]
35. Daszczuk, W.B. Discrete Event Simulation of Personal Rapid Transit (PRT) Systems. *arXiv* **2017**, arXiv:1705.05237.
36. Weitzel, D. Enabling Microsoft OneDrive Integration with HTCondor. Available online: https://arxiv.org/pdf/1907.03688.pdf (accessed on 2 March 2020).
37. Bockelman, B.; Bejar, J.C.; Hover, J. Interfacing HTCondor-CE with OpenStack. *J. Phys. Conf. Ser.* **2017**, *898*, 092021. [CrossRef]
38. Maciulaitis, R.; Brener, P.; Hampton, S.; Hildreth, M.; Hurtado Anampa, K.P.; Johnson, I.; Kankel, C.; Okraska, J.; Rodriguez Rodriguez, D.; Simko, T. Support for HTCondor high-Throughput Computing Workflows in the REANA Reusable Analysis Platform. Available online: https://cds.cern.ch/record/2696223/files/CERN-IT-2019-004.pdf (accessed on 2 March 2020).
39. Erli, G.; Fischer, F.; Fleig, G.; Giffels, M.; Hauth, T.; Quast, G.; Schnepf, M.; Heese, J.; Leppert, K.; de Pedro, J.A.; et al. On-demand provisioning of HEP compute resources on cloud sites and shared HPC centers. *J. Phys. Conf. Ser.* **2017**, *898*, 1–6. [CrossRef]
40. Asagba, P.O.; Ogheneovo, E.E. Qualities of Grid Computing that can last for Ages. *J. Appl. Sci. Environ. Manag.* **2007**, *12*, 47–52. [CrossRef]
41. Georgatos, F.; Gkamas, V.; Ilias, A.; Kouretis, G.; Varvarigos, E. A Grid-enabled CPU Scavenging Architecture and a Case Study of its Use in the Greek School Network. *J. Grid Comput.* **2010**, *8*, 61–75. [CrossRef]
42. Fu, H.; Wang, Z.; Chen, X.; Fan, X. A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques. *Softw. Qual. J.* **2018**, *26*, 855–889. [CrossRef]
43. Lima, B.; Faria, J.P. Towards the Online Testing of Distributed and Heterogeneous Systems with Extended Petri Nets. In Proceedings of the 10th International Conference on the Quality of Information and Communications Technology (QUATIC), Lisbon, Portugal, 6–9 September 2016; pp. 230–235. [CrossRef]
44. Hoare, C.A.R. *Communicating Sequential Processes*; Prentice-Hall, Inc.: Upper Saddle River, NJ, USA, 1985. [CrossRef]
45. Fonseca i Casas, P. Transforming classic Discrete Event System Specification models to Specification and Description Language. *Simulation* **2015**, *91*, 249–264. [CrossRef]
46. Daszczuk, W.B. Integrated Model of Distributed Systems. Available online: https://www.springer.com/gp/book/9783030128340 (accessed on 2 March 2020).
47. Mazzanti, F.; Ferrari, A.; Spagnolo, G.O. Towards formal methods diversity in railways: An experience report with seven frameworks. *Int. J. Softw. Tools Technol. Transf.* **2018**, *20*, 263–288. [CrossRef]
48. Karna, A.; Chen, Y.; Yu, H.; Zhong, H.; Zhao, J. The role of model checking in software engineering. *Front. Comput. Sci.* **2018**, *12*, 642–668. [CrossRef]
49. Yousefian, R.; Rafe, V.; Rahmani, M. A heuristic solution for model checking graph transformation systems. *Appl. Soft Comput.* **2014**, *24*, 169–180. [CrossRef]
50. Pira, E.; Rafe, V.; Nikanjam, A. Deadlock detection in complex software systems specified through graph transformation using Bayesian optimization algorithm. *J. Syst. Softw.* **2017**, *131*, 181–200. [CrossRef]

51. HTCondor User's Manual. Managing a Job. Available online: https://htcondor.readthedocs.io/en/stable/users-manual/managing-a-job.html (accessed on 2 March 2020).
52. Stevens, W.R.; Rago, S.A. Advanced Programming in the UNIX Environment. Available online: http://ptgmedia.pearsoncmg.com/images/9780321637734/samplepages/0321637739.pdf (accessed on 2 March 2020).
53. Müller, J.; Lorenz, M.; Geller, F.; Zeier, A.; Plattner, H. Assessment of communication protocols in the EPC Network—Replacing textual SOAP and XML with binary google protocol buffers encoding. In Proceedings of the IEEE 17Th International Conference on Industrial Engineering and Engineering Management, Xiamen, China, 29–31 October 2010; pp. 404–409. [CrossRef]
54. Eaddy, M. C# versus Java. Available online: http://www1.cs.columbia.edu/~eaddy/publications/csharpvsjava-eaddy-ddj-feb01.pdf (accessed on 2 March 2020).
55. Hericko, M.; Juric, M.B.; Rozman, I.; Beloglavec, S.; Zivkovic, A. Object Serialization Analysis and Comparison in Java and .NET. *SIGPLAN Not.* **2003**, *38*, 44–54. [CrossRef]
56. Cui, Y. Performance Test—BinaryFormatter vs Protobuf-Net. Available online: https://theburningmonk.com/2011/08/performance-test-binaryformatter-vs-protobuf-net/ (accessed on 2 March 2020).
57. Feldmeier, D.C. A Survey of High Performance Protocol Implementation Techniques. Available online: https://link.springer.com/chapter/10.1007/978-1-4615-3194-4_2 (accessed on 2 March 2020).
58. Fayeez, A.; Keedwell, E.; Collett, M. H-ACO: A Heterogeneous Ant Colony Optimization Approach with Application to the Travelling Salesman Problem. In *Artificial Evolution*; Lutton, E., Legrand, P., Parrend, P., Monmarché, N., Schoenauer, M., Eds.; Springer International Publishing: Paris, France, 2017; pp. 144–161. [CrossRef]
59. Mono-Project Main Webpage. Available online: https://www.mono-project.com/ (accessed on 2 March 2020).
60. Shapp Library Repository. Available online: https://github.com/saleph/shapp (accessed on 2 March 2020).