Leveraging K8s to implement PARCS.NET

Oleksii Bohusevych[1] and Oleksandr Derevianchenko[1]

[1]*Taras Shevchenko National University of Kyiv, Volodymirska str., 60, Kyiv, 01601, Ukraine*

Abstract

An approach to implementing PARCS (Parallel Asynchronous Recursive Control System) based on the Kubernetes container orchestration technology is presented. Its automated deployment is then exemplified by employing Microsoft Azure, a platform providing a managed solution for the technology's infrastructure. Finally, the system is utilized to tackle an applicable mathematical problem, demonstrating the proposed framework's ease of use, speed, and effectiveness.

Keywords [1]

Cloud technologies, Distributed computing, PARCS, Kubernetes

1. Introduction

There are two ways to introduce parallelism into programming languages. The first is to build basic parallel programming features in C++, Java, C#, and Python. The second approach is constructing unique add-ons over standard procedural languages: MPI, OpenMP, Cuda, OpenCL, and JavaCL. We propose PARCS as a universal add-on extension to the base programming languages.

PARCS technology's key idea is to represent a complex parallel process as execution activities that develop and communicate in some logically connected space. Current activities of a process are assigned to some space coordinates that define communication addresses. We call such a communication structure Control Space (CS). Structurally, CS consists of addressable points and channels. Both points and channels could be constants or variables. The ends of a channel are points, and points are connected by channels. Transmitting data or other information is done through channels exclusively.

Another conceptual feature of the PARCS technology is an Algorithmic Module (AM), a program in a base language extended by special commands for interacting with CS. Thus, CS can be viewed as a parallel process tree containing executing AMs at the vertices of this tree.

Topologically, CS can be viewed as a dynamically changeable graph. The fulfillment of AM can modify the structure of CS: attaching or deleting points and channels, blocking some processes, and creating and destroying AM. Independent AMs could be executed in parallel. Theoretically, CS points can include control subspaces of an arbitrary depth. AMs located in points can interchange messages through connecting channels.

As seen, the PARCS concept generalizes the concept of communicating sequential processes considered by E. Dijkstra [7] and developed by C.A.R. Hoare [8] and P. Brinch Hansen [9].

So far, there's been numerous instances of the PARCS model being implemented. Starting from the first works in 1980 [1], these varied by programming language (e.g., PASCAL, FORTRAN, C, C++, C#, Java, Python) [6], feature richness (e.g., machine-local points, intelligent task scheduling, UI), deployment model (e.g., cloud-oriented, OS-dependent), etc. PARCS-JAVA [2] proved the most popular, forming the basis for a book used by the Taras Shevchenko National University of Kyiv students. Nevertheless, only a few variations mentioned above referred to containerization techniques [12]. Yet those that did, we believe, did not exploit the technology's full potential.

Information Technology and Implementation (IT&I-2023), November 20-21, 2023, Kyiv, Ukraine
EMAIL: oleksii.bohusevych@mail.com (A. 1); drug_2002@ukr.net (A. 2)
ORCID: 0009-0009-7620-7298 (A. 1); 0009-0006-9308-8567 (A. 2)

Using containers to represent parts of a PARCS programming system provides numerous advantages. Firstly, containers are platform-agnostic, meaning they operate seamlessly across various environments. On top of that, containers are lightweight, which significantly expedites the process of horizontal scaling and feature delivery. Furthermore, containers run in complete isolation, which makes them an ideal foundation for distributed system architectures.

For the containers to integrate, an orchestration tool is usually required [13]. Such puts the applications, or workloads, in a dedicated virtual network, where they can communicate, concealed from the outside world. The orchestrator is a facilitator and mediator within the network, providing transmission abstractions and load balancing. In addition, it ensures resource availability, manages progressive deployment, handles automatic scaling, and disposes of unhealthy containers.

This paper will analyze the previous achievements in the sphere and introduce a PARCS programming system whose infrastructure solely relies on containers. The orchestration tool used is Kubernetes by Google, known for its declarative approach to defining a cluster's desired state. The K8s tool is the industry's standard, with excellent community support, open-source code, extensive documentation, diverse features, and abundant cloud service provider integrations.

The work aims to build a PARCS system based on the modern version of the .NET platform and automate its deployment.

To achieve this goal, the following tasks were carried out:

1. Analyzing previous developments in the field and systematizing existing experience.
2. Designing and developing an analog of the system using contemporary programming tools based on the .NET platform.
3. Conducting testing and ensuring the developed system's effectiveness and compliance with the classical concept of PARCS.
4. Creating means of system deployment which would require as little human intervention as possible.

2. Related work

The proposed work aims to contribute to the PARCS programming systems based on .NET, which comprise PARCS.NET (2015) [3] and PARCS-WCF (2021) [5]. Hence, these will be considered in more detail.

On the one hand, PARCS.NET was the first PARCS programming system built on top of .NET. It used NuGet packages to segregate domain-specific contracts that enabled AM development, TCP-based console applications running daemons and the host, and an OWIN-based REST API managing users, modules, and tasks. The host would initiate point creation and daemon discovery as a root of the control space hierarchy, while daemons would run and cancel jobs, execute AMs, etc.

Among the system's assets, one can distinguish independent AM development through add-on packages, a low-latency communication protocol, partial containerization, and various cloud integrations. At the same time, PARCS.NET relies on .NET Framework 4.6.1, an obsolete, no longer supported Windows-dependent technology [14]. AMs in PARCS.NET, which cannot be created programmatically but only manually placed in the system's environment, are synchronous, which may lead to thread starvation at a bigger scale. Finally, the system can only be published on-premises or as IaaS, significantly complicating maintenance and mobility.

On the other hand, PARCS-WCF was a more modern follow-up, mainly created to model distributed operations of relational choice algebra. It consisted of a WCF web service exposing TCP and NETTCP bindings and console applications describing AMs.

PARCS-WCF brought asynchronous AMs, a fallback to a different communication protocol to connect two machine-local points, secure connectivity, and customizable serialization. However, the framework under the hood remained obsolete [14], and the deployment model, which, given that the technology stack would still end up at the IaaS level at most, was missing.

3. Architecture

The proposed PARCS programming system comprises the following components (see Figure 1):

- the host – a unit for scheduling algorithmic modules and centralized control space management.
- the daemons – worker units running the actual compute-intense workloads.
- the storage – a hierarchical metadata repository holding unstructured execution logs, task inputs, and outputs.
- the database – a relational append-only repository with user information and task history.
- the portal – a graphical interface for user-friendly interaction with the system.

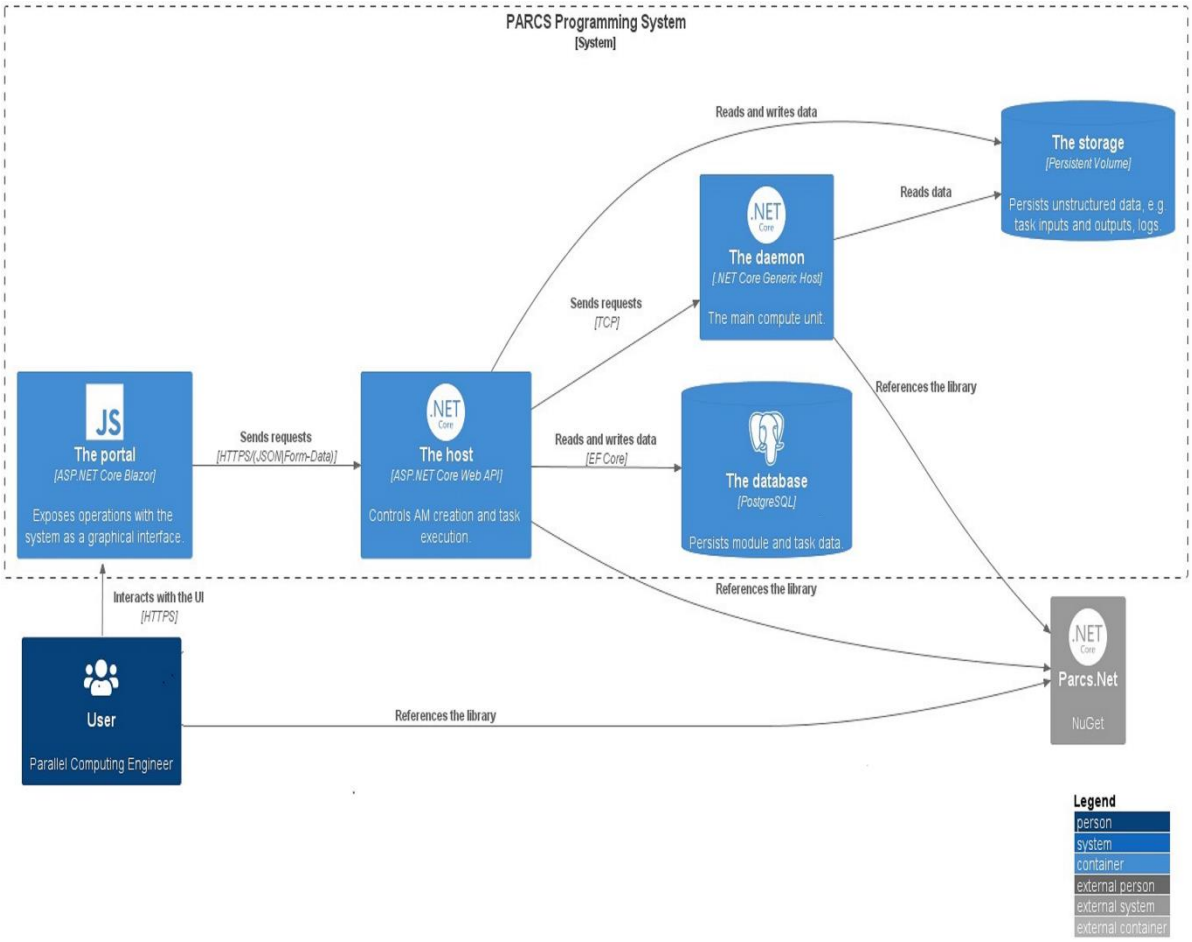

Figure 1: Proposed system architecture

PARCS models are defined through algorithmic modules (AM), which are, in fact, add-on extensions on top of base programming languages (see Figure 2), e.g., C#, Java, and C++. The modules are developed independently of the PARCS programming system using the contracts defined in a shared library (SEND DATA TO CHANNEL, ACCEPT DATA FROM CHANNEL, CREATE/DELETE POINT, CREATE/DELETE CHANNEL, ASSIGN AM TO POINT and some others). Once the modules are ready for deployment, they are compiled and programmatically submitted into the system as assemblies, where they are discovered, parametrized, and executed using isolated reflection.

PARCS portal is a Blazor-based graphical interface enabling end-user interaction with the system. It exposes such operations as programmatic module creation and disposal, task execution, cloning, deletion, repeat, and cancellation. It also allows for dynamic AM discovery and exploration of execution output and logs.

PARCS host is a web API giving access to more granular operations like task status updates, retrieval of module information, etc. It also provides two ways of running tasks: synchronously and asynchronously. The first one solely serves debugging purposes, as the caller must wait for the request to be fully processed. In comparison, the second one, enabled by the webhook URI parameter, leaves the execution to a background process, making a callback to the provided URI upon finishing.

Finally, PARCS daemons operate on a signal-based model. Every communication session with them is preceded by a signal exchange, during which daemons are informed of the payload they will receive and the action they must take.

```
public async Task RunAsync(IModuleInfo moduleInfo, CancellationToken cancellationToken = default)
{
    var moduleOptions = moduleInfo.ArgumentsProvider.Bind<ModuleOptions>();

    double a = 0;
    double b = Math.PI / 2;
    double h = moduleOptions.Precision ?? 0.00000001;

    var pointsNumber = moduleInfo.ArgumentsProvider.GetPointsNumber();
    var points = new IPoint[pointsNumber];
    var channels = new IChannel[pointsNumber];

    for (int i = 0; i < pointsNumber; ++i)
    {
        points[i] = await moduleInfo.CreatePointAsync();
        channels[i] = await points[i].CreateChannelAsync();
        await points[i].ExecuteClassAsync<WorkerModule>();
    }

    double y = a;
    for (int i = 0; i < pointsNumber; ++i)
    {
        await channels[i].WriteDataAsync(y);
        await channels[i].WriteDataAsync(y + (b - a) / pointsNumber);
        await channels[i].WriteDataAsync(h);
        y += (b - a) / pointsNumber;
    }
    DateTime time = DateTime.Now;
    Console.WriteLine("Waiting for result...");

    double result = 0;
    for (int i = pointsNumber - 1; i >= 0; --i)
    {
        result += await channels[i].ReadDoubleAsync();
    }

    Console.WriteLine("Result found: res = {0}, time = {1}", result, Math.Round((DateTime.Now - time).TotalSeconds, 3));

    var bytes = Encoding.UTF8.GetBytes(result.ToString());
    await moduleInfo.OutputWriter.WriteToFileAsync(bytes, moduleOptions.OutputFilename);

    for (int i = 0; i < pointsNumber; ++i)
    {
        await points[i].DeleteAsync();
    }
}
```

1

2

3

4

Figure 2: An example of an algorithmic module in the proposed system. Here, the first box denotes control space initialization, the second box – the fulfillment of the space, the third box – results collection, and the fourth – control space disposal.

AM execution is done in a separate load context, not interfering with the dependencies of the hosting process. Depending on the size and complexity of the run PARCS model, a single task initiation may result in thousands of points and commutations created. To dispose of those gracefully in case of unresolvable error, the host sends an appropriate signal to all the live daemons. Upon receiving such, each daemon checks the failed task identifier against the tasks it executes and terminates the task if there is a match. For possible investigations later, the error logs persist in the storage.

4. Platform

From the .NET side of things, as opposed to the earlier mentioned obsolete frameworks, the solution components run on the seventh, the newest version of the platform as of autumn 2023. Compared to the versions used in the two older counterparts, .NET 7 is open-source and modular, has various cloud integrations and support, a comprehensive development toolkit, and numerous security enhancements tackling up-to-date vulnerabilities. It is also significantly more performant, as many benchmarks suggest [15]. Most importantly, .NET 7 is cross-platform [11], meaning its target infrastructure is OS-

independent. This, among other things, enables extensive containerization and broadens the range of environments the system can be deployed to.

Newer .NET versions also have some unique features that proved helpful when building the system. One such is Channels [16][16], a set of synchronization data structures for passing data between producers and consumers within a process. Their programming model for data exchange suggests asynchronous communication, which fits perfectly in the PARCS concept, as it is the letter 'A' in the acronym. The only nuance is that commutations within PARCS are bidirectional, unlike those in the publish/subscribe model. To tackle that, a pair of unidirectional channels can be configured for each pair of connected points that reside in the same process, effectively enabling efficient process-local communication, a feature first introduced in PARCS-WCF.

Assembly Load Context [17], a runtime provider for locating and loading dependencies, is another feature exclusively available in .NET Core and .NET 5+ applications. It helps load and unload code dynamically by creating an isolated environment for managed assemblies and their dependencies. Utilizing it massively facilitated the making of a plugin-oriented architecture around algorithmic modules.

5. Integration

The topic of containerizing PARCS, described in [4], was further developed here. In the first place, this direction was elaborated on because of the degree of automation containers enable. Containers are way more portable and independent than virtual machines used previously because of the absence of the operational system layer. Moreover, containerized applications are published along with their dependencies, making the applications run predictably across different environments.

Still, the missing OS layer in the container architecture has its disadvantages. First, giving administrative privileges to container processes is undesirable since the technology has direct access to kernel subsystems, which can lead to vulnerability exploitations and the whole system being compromised. Second, containers are built to be run on machines with specific OS, i.e., Linux or Windows. Unfortunately, Linux-oriented containers were previously impossible due to the used .NET framework constraints. Luckily, this barrier has been overcome via the platform upgrade introduced.

```
3   FROM mcr.microsoft.com/dotnet/runtime:7.0 AS base
4   WORKDIR /app
5
6   FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build
7   WORKDIR /src
8   COPY ["src/Parcs.Daemon/Parcs.Daemon.csproj", "src/Parcs.Daemon/"]
9   COPY ["src/Parcs.Core/Parcs.Core.csproj", "src/Parcs.Core/"]
10  RUN dotnet restore "src/Parcs.Daemon/Parcs.Daemon.csproj"
11  COPY . .
12  WORKDIR "/src/src/Parcs.Daemon"
13  RUN dotnet build "Parcs.Daemon.csproj" -c Release -o /app/build
14
15  FROM build AS publish
16  RUN dotnet publish "Parcs.Daemon.csproj" -c Release -o /app/publish /p:UseAppHost=false
17
18  FROM base AS final
19  WORKDIR /app
20  COPY --from=publish /app/publish .
21  ENTRYPOINT ["dotnet", "Parcs.Daemon.dll"]
```

Figure 3: Daemon Docker file

Figure 3 depicts a Docker file, i.e., instructions for creating a Docker image to run the daemon. As can be seen from it, the build kind is multistage. First, the .NET runtime is marked as the base image (lines 3-4). After that, the source code of the Parcs.Daemon and Parcs.Core projects is copied to the working directory. Their dependencies are resolved, and the code is compiled using the optimized Release configuration (lines 6-13). Next, the application is published (lines 15-16), and the resulting

binaries are copied to the main directory (lines 18-21). It is also worth noting that the base *mcr.microsoft.com/dotnet/runtime:7.0* image has only the code and libraries needed for the program to work. At the same time, mcr.microsoft.com/dotnet/sdk:7.0, which is used for compiling and deploying the projects, is a complete software development kit. This segregation demonstrates the power of multistage builds, resulting in smaller and faster images.

Similarly, the other parts of the solution, i.e., the host and the portal, are containerized. For those components to form a single functional system, their integration must be ensured. At the very least, the portal needs a way to communicate with the host, while the host should be able to discover daemons. To achieve that, the naïve approach would expose container ports externally and reference those at the application level. However, doing so is undesirable since such public access weakens security by enlarging the attack surface. Moreover, this limits scaling, as containers are now explicit rather than abstract: adding a new instance forces configuration changes and redeployment.

Because of the reasons mentioned above, employing a managed solution for container orchestration is highly recommended. Some benefits orchestration introduces include automated deployment and scaling, monitoring and logging, intelligent service discovery and optimized load balancing, role-based access control, high availability, and redundancy.

One can distinguish Kubernetes, Docker Compose, and Mesos among the orchestration tools. Thanks to the Visual Studio integration, Docker Compose is relatively straightforward to start working with. With the orchestration support added to a .NET solution, Visual Studio automatically generates YAML files with cluster definition, provisioning which is effectively a matter of one command: "docker-compose up." Albeit straightforward, Docker Compose is feature-limited and is often used during the development phase.

Kubernetes [18], on the other hand, is production ready. Hence, it was eventually chosen to orchestrate live containers in the considered architecture. One distinctive feature of this technology is its declarative, rather than imperative, approach to defining the desired state of the environment: once the state is described and published, Kubernetes is responsible for changing the current one to match the desired one at a controlled rate.

For the proposed system, the following Kubernetes components are declared:

- Deployments:
    - Parcs-Daemon
    - Parcs-Host
    - Parcs-Portal
    - Parcs-Database
- Services:
    - Parcs-Daemon (headless)
    - Parcs-Host (cluster IP)
    - Parcs-Portal (load balancer)
    - Parcs-Database (cluster IP)
- Persistent volume claims:
    - Parcs-Storage
    - Parcs-Database

So-called deployments are workload resources that represent actual applications hosted as containers in abstract structures called pods, which, in turn, are organized into nodes. Within the PARCS terminology, such hierarchy enables three-level scaling: points may be created within a single daemon or multiple, stretching over containers, pods, and nodes.

So-called services [19] are intermediaries facilitating communication within the cluster. They can be of four types: Cluster IP, Node Port, Load Balancer, and External Name. Cluster IP is the default, which exposes a cluster-internal static IP address; Load Balancer is designed for external gateway scenarios, as it provides an entry point into the cluster; Node Ports allocate a custom port; and, finally, External Name defines mappings to CNAME records.

Since Parcs-Host is meant to be consumed only by Parcs-Portal, i.e., internally, the type for the service to target it is Cluster IP. Similarly, Parcs-Database, only visible to Parcs-Host, also gets Cluster IP. In contrast, Parcs-Portal is user-facing, so its service type should be Load Balancer.

There is another type to add to the list: Headless. It does not have a static IP address or a port assigned. Its primary purpose is to serve scenarios where application discovery is done without being

tied to Kubernetes' implementation. In such cases, A and AAAA records for the pods hosting the workloads targeted by the service are automatically added to the DNS configuration of the cluster. Whenever there is a need to discover all such workloads, a request to the cluster's DNS server is made, and the list of the corresponding IP addresses is returned. This exact mechanism was utilized to implement the discovery of Parcs-Daemons, dynamic applications that needed to be communicated directly by Parcs-Host and themselves upon request.

Given the service abstractions mentioned above, it is hard to argue that a move away from the traditional PARCS techniques around accessing daemons directly could be made. As such, communication between the host and them could go through a Kubernetes service, balancing the load and choosing the best request handler based on cluster heuristics like CPU utilization and memory usage. Another possibility could be internally employing the Kubernetes API, which provides the same data. As a result, advances and further research in this direction are yet to be made.

Figure 4: The Kubernetes definition of the daemon.

Figure 4 exemplifies a partial definition of the Kubernetes cluster, namely the daemon definition. The first and fourth boxes signify two integral parts of the Kubernetes daemon, i.e., the service and deployment. The service exposes a port for external traffic and forwards it to the respective port on the target workload pods, as denoted by the second box. The configuration marked by the third box removes the service's static IP address so that target pods can be discovered dynamically. The fifth box represents the workload's desired number of container instances, while the sixth box points to the repository from which the container image is expected to be pulled. Lastly, box 8 showcases how metadata storage is mounted into the daemon containers' filesystem.

6. Networking

When two connected points reside in different processes, whether on the same node or not, message-oriented information exchange in the proposed PARCS programming system happens over TCP, a low-overhead transport-level networking protocol. However, this protocol does not respect message boundaries [20], meaning not every Receive statement on the server side corresponds to a single Send statement on the client side. This is because of the logic behind the sockets' Receive method. Instead of reading messages sequentially as they arrive in the TCP buffer, the method pulls all the available data and reads it all in one go.

To tackle this problem, there are three possibilities:

- Send fixed-size messages.
- Send the message size with each message.
- Use a marker system to separate messages.

The first approach is suboptimal regarding result volumes, as clients will be forced to send excessive data. The third would require devising an intricate system of terminating characters. Albeit trivial, the second solution was guaranteed to be efficient. Hence, it was opted for. Thus, every time a daemon or the host received a transmission, they would consider the first bytes as an integer indicating the size of the follow-up message and then poll the TCP buffer until the size is exhausted.

7. Deployment

Kubernetes, known as K8s, is an open-source system for managing containerized applications across multiple hosts. It provides basic mechanisms for deploying, maintaining, and scaling applications [21]. It is a full-fledged, standalone, independent technology that can be deployed to a dedicated server or onto a cloud platform. The former allows for greater control over the software, which comes at the cost of maintenance overhead, while the latter is more cost-effective, resilient, robust, available, and secure. One can distinguish Azure Kubernetes Service [22] (AKS) by Microsoft, Kubernetes on AWS [23] by Amazon, and Kubernetes Engine by Google [24] among Kubernetes-ready most common out-of-box PaaS. Since tweaks to enable this or that integration are negligible, and the features offered are similar, Microsoft Azure was chosen for demonstration purposes.

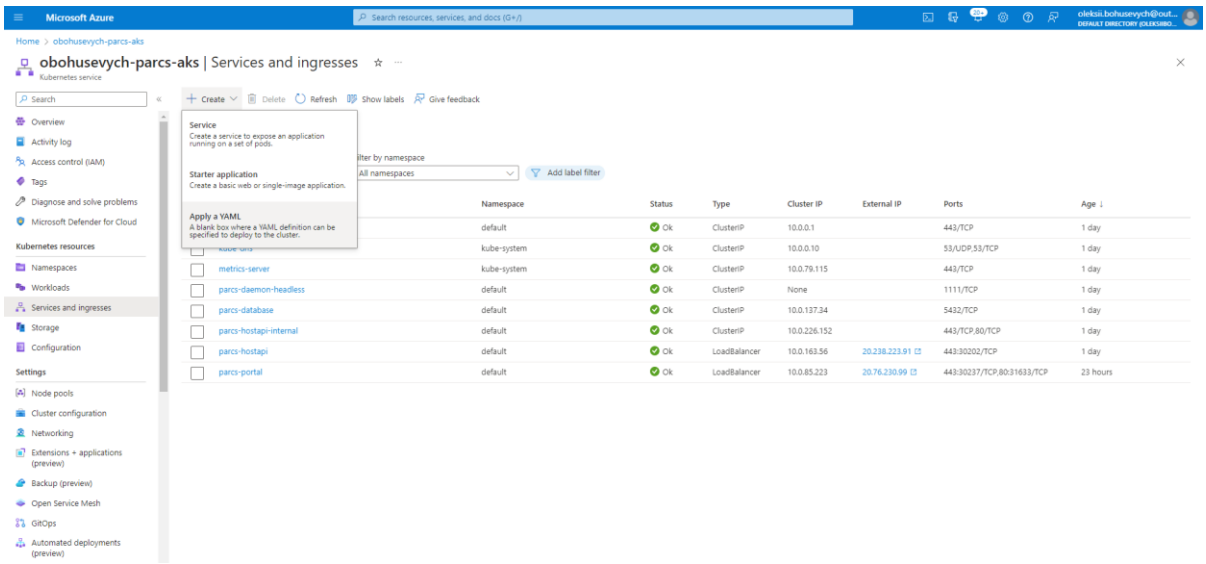

Figure 5: Kubernetes-based PARCS deployed to the Azure cloud as AKS.

Creating an instance of AKS (see Figure 5) can be achieved in many ways [25]: manually through the portal, using PowerShell/Bash scripts, via Infrastructure as Code (IaC) tools like Terraform [26] or Bicep [27], or through in-built ARM templates. Understandably, the former two options do not allow for much automation, so only the rest was considered. Among those considered, ARM templates stand

out as a natural part of the Azure ecosystem. However, even though they integrate seamlessly, the structure of ARM templates is counter-intuitive and overwhelming, so they were not opted for. So, Bicep and Terraform, modular, maintainable, and declarative, took precedence. Specifically, Bicep-generated ARM templates were used in this case. Once the resource was created, loading the YAML state file was the only remaining bit to finish the system's readiness, making it straightforward to start working with the PARCS technology. Logging, authentication, authorization, pool resizing, and private image repository integrations could also be configured upon request.

8. Testing

The proof of work task [6] was attempted to challenge the system's correctness and measure its performance. Proof of work is a fundamental concept of the Bitcoin cryptocurrency. It is also widely used in other environments where one party (the prover) must prove to others (the verifiers) that a certain amount of a specific computational effort has been expended.

One of the approaches to implementing the concept is based on unidirectional hash functions, e.g., SHA-256: the verifier passes arbitrary text and a number to the prover, after which the prover needs to find such a combination of symbols so that together with the original text, they give a hash with a predefined number of zeros at the start of it. Naturally, the only practical solution is the brute force method: the prover tries different options sequentially. Once the desired hash function input is found, the verifier can prove its correctness immediately by feeding it into the function.

The work, i.e., the search in the space of possible text combinations, can be done in parallel. For that, sequential chunks from that space are given to each of the workers for processing. Attempting the combinations continues until one of the workers finds a desired hash. These were the exact implementation details for this task using the proposed PARCS programming system. The results are laid out in Table 1.

The hardware used for benchmarking was provided by Azure as part of the AKS solution and consisted of virtual machines of size Standard_DS2_v2 (2 vCPU, 7 GiB RAM, 8000 IOPS/64 (86) MBps, 6400 IOPS/96 MBps). The software was packaged in the Release mode, with the runtime optimizer turned on and no debugging information included.

Test cases denote the complexity of the task, i.e., the number of zeros at the start of the desired hash function output, e.g., '000006bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad' satisfies the task of complexity 5.

Table 1
Test results for the execution of the Proof of Work PARCS model in Azure

Test Case	Sequential	Parallel (2 Nodes)	Parallel (4 Nodes)
5	0.21	0.28	0.36
6	85.93	71.24	40.08
7	445.27	405.99	251.66
8	11362.09	7848.09	4961.45

As seen from the results in Table 1, the sequential algorithm performs slightly better at first but then degrades notably as the number of trials grows. This can be attributed to the initial overhead of data distribution, omitted by the sequential variation.

9. Conclusions

The built PARCS.NET-K8s programming system encompasses the assets its historical counterparts used to have and some improvements in portability and deployment automation enabled by the contemporary technological stack. The test results demonstrate its effectiveness and operability in distributed computations.

The system is believed to have not yet discovered prospects concerning further exploration and exploitation of the containerized environment based on Kubernetes. Such features as constraining the AM execution environment, introducing authentication and authorization, intelligent load balancing, and resource allocation might be of great value.

10.References

[1] Glushkov, V.M. and Anisimov, A.V. Controlling Spaces in Asynchronous Parallel Computations, 1980. Cybernetics, 5, 1-9.

[2] Anisimov, A.V. and Derevianchenko, A.V. The System PARCS-JAVA for Parallel Computations on Computer Networks, 2005. Cybernetics and Systems Analysis, 41, 17-26. https://doi.org/10.1007/s10559-005-0037-4.

[3] Derevianchenko, O.V. and Havro, A.U. The application of PARCS. NET system and Amazon EC2 for cloud computing, 2015. Bulletin of Taras Shevchenko National University of Kyiv, Ser. Physics & Mathematics, No. 4, 111–118.

[4] Anisimov, A. V., Derevianchenko, O. V. and Khavro, A. U. The applying of PARCS.NET system with Docker containers and Google Cloud Platform for distributed cloud computing, 2018. Artificial Intelligence, No. 3(81), 52–61.

[5] Anisimov, A.V. and Fedorus, O.M. Development and Prospects of the PARCS-WCF System, 2020. Cybernetics and Systems Analysis, 56, 152-158.

[6] Anisimov A. V., Derevianchenko O. V., Kuliabko P. P. and Fedorus O. M. PARCS Technology: Concept and Implementations, 2023. Cybernetics and Systems Analysis, 59, 832–843. https://doi.org/10.1007/s10559-023-00619-6.

[7] Dijkstra, E.W. (1968) Cooperating Sequential Processes. In: Hansen, P.B., Ed., The Origin of Concurrent Programming, Springer, New York, 65-138. https://doi.org/10.1007/978-1-4757-3472-0_2.

[8] Hoare, C.A.R. (1978) Communicating Sequential Processes. Communications of the ACM, 21, 666-677. https://doi.org/10.1145/359576.359585.

[9] Brinch Hansen, P. (1978) Distributed Processes: A Concurrent Programming Concept. Communications of the ACM, 11, 934-941. https://doi.org/10.1145/359642.359651

[10] M. Jakobsson, Proofs of Work and Bread Pudding Protocols, 1999. Secure Information Networks: Communications and Multimedia Security. Kluwer Academic Publishers: 258–272.

[11] M. Price, C# 11 and .NET 7 – Modern Cross-Platform Development Fundamentals, 2022.

[12] Docker, Use containers to Build, Share, and Run your applications, 2023. URL: https://www.docker.com/resources/what-container/.

[13] IBM, What is container orchestration, 2023. URL: https://www.ibm.com/topics/container-orchestration.

[14] Microsoft, Microsoft .NET Framework Lifecycle, 2023. URL: https://learn.microsoft.com/en-us/lifecycle/products/microsoft-net-framework.

[15] D. Callan, .NET Framework 4.8 v .NET 7 reflection performance benchmarks, 2023. URL: https://davecallan.com/dotnet-framework-48-v-dotnet7-reflection-performance-benchmarks/.

[16] S. Toub, An Introduction to System. Threading. Channels, 2019. URL: https://devblogs.microsoft.com/dotnet/an-introduction-to-system-threading-channels/.

[17] J. Bytes, Dynamically Loading Types in .NET Core with a Custom Assembly Load Context, 2020. URL: https://jeremybytes.blogspot.com/2020/01/dynamically-loading-types-in-net-core.html.

[18] Kubernetes, Overview, 2023. URL: https://kubernetes.io/docs/concepts/overview/.

[19] Kubernetes, Service, 2023. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[20] A. Talazar, Solution for TCP/IP client socket message boundary problem, 2005. URL: https://www.codeproject.com/Articles/11922/Solution-for-TCP-IP-client-socket-message-boundary.

[21] Kubernetes (K8s), 2023. URL: https://github.com/kubernetes/kubernetes.

[22] Microsoft, What is Azure Kubernetes Service, 2023. URL: https://learn.microsoft.com/en-us/azure/aks/intro-kubernetes.

[23] Amazon, What is Amazon EKS, 2023. URL: https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html.
[24] Google, Google Kubernetes Engine (GKE), 2023. URL: https://cloud.google.com/kubernetes-engine.
[25] Microsoft, Quickstart: Deploy an Azure Kubernetes Service (AKS) cluster using Azure CLI, 2023. URL: https://learn.microsoft.com/en-us/azure/aks/learn/quick-kubernetes-deploy-cli.
[26] Microsoft, What is infrastructure as code (IaC), 2023. URL: https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code.
[27] Microsoft, What is Bicep, 2023. URL: https://learn.microsoft.com/en-us/azure/azure-resource-manager/bicep/overview?tabs=bicep.