Архитектура распределенной системы для потоковых вычислений с контейнеризацией и приоритизацией задач *

А. М. Соколов, А. А. Ларионов, В. М. Вишневский, А. А. Мухтаров

Институт проблем управления им. В. А. Трапезникова РАН, Москва, Россия

Аннотация. В статье представлена распределенная система для организации потоковых вычислений. Система включает в себя сервер для управления данными, управляющий сервис (супервизор), набор узлов-рабочих, на которых производится выполнение задач, и базу данных. Для абстрагирования от конкретных языков программирования и инструментов, используемых при вычислениях, реализации алгоритмов (задачи) упаковываются и выполняются в контейнерах Docker. Для эффективной работы при высокой нагрузке система поддерживает несколько стратегий приоритизации задач. Для работы с системой пользователю достаточно построить образ docker-контейнера, описать набор входных данных в JSON-файле и загрузить их через веб-интерфейс. Система может быть развернута в любом общедоступном облаке. В статье подробно описана архитектура системы и приведены численные результаты, полученные при вычислениях на различных облачных и локальных платформах. В работе изучено влияние различных стратегий приоритизации на длительность вычислений при умеренной нагрузке.

Ключевые слова: потоковые вычисления, контейнеризация, тонкая виртуализация, облачные вычисления.

DOI 10.14357/20718632230401 **EDN** HSOYNI

Введение

В ходе прикладных научных исследований часто возникает необходимость получить численные результаты применения математических моделей или алгоритмов на больших объемах входных данных. При этом модели и алгоритмы могут быть реализованы на разных языках программирования или использовать различные программные инструменты, такие как GNU Octave, NS-3, OMNeT++ и др. Особенностью таких задач является то, что результаты для каждого набора входных данных можно получать независимо, то есть при расчетах на разных входных наборах не требуется синхронизация. Поэтому сократить время получения результатов можно с помощью горизонтального масштабирования, проводя расчеты одновременно на множестве рабочих узлов. Для автоматизации проведения подобных расчетов в данной статье предлагается использовать систему для потоковых вычислений, в которой алгоритмы расчетов передаются и выполняются в docker-контейнерах, а для эффективной обработки потока задач, поступающего от множества пользователей, используются различные стратегии приоритизации. Применение контейнеризации позволяет

* Работа выполнена при поддержке Российского научного фонда (22-49-02023).

абстрагироваться от используемых языков и инструментов научных вычислений, а наличие приоритизации позволяет эффективно распределить время между множеством пользователей и задач. Предлагаемая система представляет собой веб-приложение с микросервисной архитектурой и может быть развернуто в любом публичном облаке.

Система для потоковых вычислений, представленная в данной статье, уже использовалась авторами при проведении ряда исследований в области теории массового обслуживания и моделирования телекоммуникационных сетей. Например, в работе [1] использовалась аппроксимация характеристик сложной системы массового обслуживания с помощью нейронных сетей, для обучения которых требовалось создать большой синтетический набор входных данных. Каждый элемент этого набора включал параметры системы массового обслуживания и значения ее характеристик, которые вычислялись с помощью имитационной модели. Весь набор входных данных состоял из 500 тысяч записей, а каждое выполнение имитационной модели в среднем занимало порядка 5 секунд. Использование предлагаемой системы потоковых вычислений позволило получить численные результаты менее чем за сутки, не требуя изменений кода однопоточной имитационной модели или написания каких-либо специальных скриптов для распараллеливания вычислений.

Существует множество систем для организации распределенных и потоковых вычислений. Наиболее близкими к системе, предлагаемой в этой статье, являются системы для добровольных вычислений (volunteer computing) [2-10], в которых вычислительные мощности предоставляются многими пользователями, и системы для локальных распределенных вычислений [11-13].

В ряде систем для добровольных вычислений для работы пользователю необходимо использовать специальные библиотеки [5-10]. Так, в системе BIONIC (Berkeley Open Infrastructure for Network Computing) [5-7] пользователь устанавливает на компьютер SDK, определяет предоставляемые ресурсы, и сам выбирает проект, задачи которого готов взять в работу. Так как вычисления могут проводиться на разных платформах, разработчикам приходится поддерживать различные версии программ под разные операционные системы и архитектуры. Специальный SDK используется и в других системах, например в системе Golem [8]. Отметим, что Golem, как и некоторые другие системы добровольных вычислений (например, VFuse [9]) позволяет организовать взаиморасчеты между потребителями и провайдерами вычислительных мощностей с помощью технологии блокчейн. На момент написания статьи в системе Golem было доступно 11628 ядер на 620 серверах. Отличительной особенностью проекта [10] является использование в качестве языка программирования для расчетов JavaScript, применение технологии Web Workers для запуска нескольких потоков и выполнения расчетов в браузерах на подключенных к системе компьютерах.

Существует большое количество систем, использующихся для научных расчетов локально без предоставления мощностей для публичного использования. В качестве примера можно привести системы Everest [11] и Nimrod [12]. Отличительной особенностью системы Nimrod является наличие декларативного языка для описания задач, а также возможность установления связи между задачами и использование результатов одних задач в качестве входа для других. В статье [13] рассматривается система для вычисления трудоемких задач JINR, в которой для управления ресурсами используется технология OpenNebula. Основным преимуществом при использовании технологии OpenNebula является объединение различных виртуальных машин, работающие с разными технологиями виртуализации (VMware, KVM, LXD, Firecracker) в единый кластер для вычислений.

Отдельно стоит отметить систему Worldwide LHC Computing Grid [14], используемую для расчета задач в области ядерной физики институтом CERN, содержащую около 1,4 млн ядер и 1,5 эксабайт памяти, распределенных по 42 странам. Основная задача комплекса заключается в предоставлении ресурсов для хранения, распространения и анализа данных, получаемых в экспериментах на Большом адронном коллайдере.

Предлагаемая в настоящей статье система для потоковых вычислений имеет ряд отличий от существующих систем. Во-первых, при реализации вычислительных алгоритмов не требуется

использовать какие-либо специальные библиотеки, а применение контейнеризации позволяет эффективно изолировать окружение, в котором выполняются задачи, от операционной системы узла-рабочего, требуя меньше ресурсов по сравнению с традиционной виртуализацией. Во-вторых, в предлагаемой системе задачи при выполнении никак не связаны и не синхронизируются. Более того, порядок выполнения задач внутри одного задания никак не определен заранее. Соответственно, не требуется использовать библиотеки для межпроцессного взаимодействия наподобие MPI. Наконец, в-третьих, приложение предназначено для небольших исследовательских задач и моделирования относительно небольших систем, для вычисления которых не требуются мощности сотен тысяч компьютеров или суперкомпьютера, как в GRID.

1. Архитектура системы

По своей структуре система для параллельных расчетов является микросервисным веб-приложением, которое состоит из нескольких базовых сервисов, показанных на Рис. 1. Пользователь взаимодействует с системой с помощью веб-интерфейса или с помощью REST API. Бэкенд — это серверная часть приложения для управления и мониторинга задачами, авторизации пользователей и сбора статистики выполнения задач. Супервизор используется для управления рабочими и мониторинга очереди задач. Рабочие выполняют подзадачи, создавая и запуская контейнеры из переданных пользователем Docker-образов. Каждая подзадача выполняется в отдельном контейнере.

Система использует две базы данных: реляционную БД Postgres и Redis. В отличие от Postgres, данные Redis хранятся в оперативной памяти, что обеспечивает быстрый доступ, но создает проблемы для длительного хранения. По этой причине Redis используется для хранения данных, актуальных только во время выполнения сервиса, а также для взаимодействия между компонентами, а Postgres - для хранения параметров задач, результатов вычислений, профилей пользователей и прочих данных, доступ к которым нужно иметь постоянно.

Введем определение терминов “задача” и “подзадача”, которыми будем пользоваться далее в статье. **Задача** характеризуется алгоритмом ее решения, набором параметров и массивом входных данных. Программная реализация алгоритма содержится в docker-образе. Система автоматически декомпозирует задачу на маленькие части, которые называются **подзадачами**. Каждая подзадача характеризуется отдельным набором входных данных.

1.1. Задачи: определение, жизненный цикл, приоритизация

Рассмотрим жизненный цикл задачи и подзадачи. После загрузки пользователем файла с задачей, она декомпозируется на подзадачи. И задача, и все ее подзадачи в этот момент находятся в состоянии INITIALIZED. После того, как

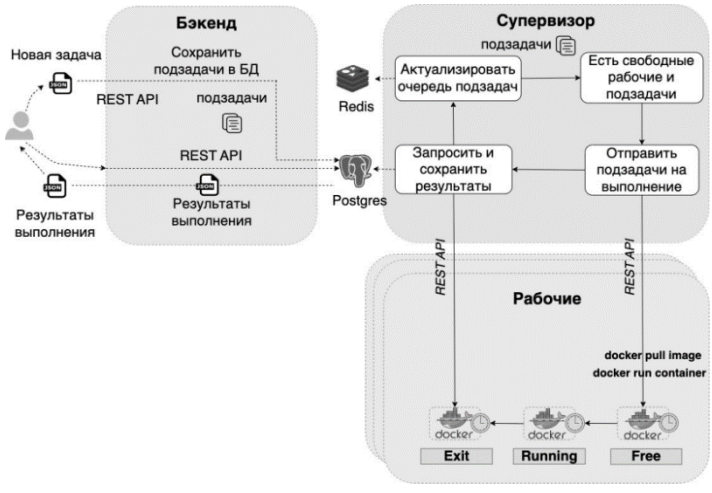

Рис. 1. Архитектура системы для параллельных вычислений

пользователь отправил задачу на исполнение, она переходит в состояние RUNNING. Подзадачи поочередно добавляются в очередь в соответствии с их приоритетом, переходя в состояние ST-QUEUED. Когда освободившийся рабочий берет подзадачу на исполнение, она переходит в состояние ST-RUNNING, откуда переходит либо в ST-COMPLETED при удачном завершении, либо в ST-ERROR при ошибке. Когда все подзадачи завершились, задача покидает состояние RUNNING. Если все подзадачи оказались в состоянии ST-COMPLETED, то задача переходит в состояние COMPLETED, а если хотя бы одна подзадача завершилась с ошибкой и оказалась в ST-ERROR, то и вся задача переходит в состояние ERROR. Также выполнение задачи и всех оставшихся подзадач можно поставить на паузу, изменяя состояние задачи с RUNNING на STOPPED. При этом все выполняющиеся или ожидающие в очереди подзадачи отменяются и возвращаются в состояние INITIALIZED. Для того, чтобы некоторые задачи были вычислены быстрее других, пользователи могут назначать задачам приоритеты, от первого (самого высокого) до пятого (самого низкого). Система поддерживает четыре стратегии приоритизации: **FIFO** - задача, поступившая в систему раньше, выполняется раньше. В этом режиме приоритет не имеет значения; **UNIFORM** - рабочие поровну делятся между выполняющимися задачами. Если рабочих меньше, чем задач, рабочие берут более ранние задачи. Количество одновременно выполняющихся задач никогда не превышает числа рабочих; **PRIORITY FIFO** - задачи сортируются по приоритету и времени поступления. Если в момент поступления задачи более высокого приоритета в системе выполнялась менее приоритетная задача, то выполнение последней приостанавливается; **PRIORITY UNIFORM** - задачи с наивысшим приоритетом вычисляются первыми, рабочие распределяются поровну между задачами с одинаковым приоритетом. Если в момент поступления задачи более высокого приоритета в системе выполнялась менее приоритетная задача, новые подзадачи последней не ставятся на выполнение до окончания расчета всех приоритетных задач.

1.2. Супервизор

Супервизор управляет выполнением задач, распределяя подзадачи между рабочими. Сервис отслеживает состояние рабочих, степень заполнения буфера подзадач, передает подзадачи из буфера на выполнение рабочим по мере их освобождения, записывает результаты расчета в базу данных Postgres, а также останавливает выполнение при получении команды от бэкенда. Сервис реализован на языке Python.

Для взаимодействия с бэкендом супервизор использует Redis. Бэкенд сообщает супервизору о появлении новой задачи или запрашивает приостановку выполнения уже выполняющейся задачи в случае получения команды STOP или появления более приоритетной задачи. Также в Redis хранится состояние буфера, в котором хранятся подзадачи, которые будут назначены рабочим при их освобождении. Подзадачи в буфере отсортированы в соответствии с выбранной стратегией приоритизации. Когда очередной рабочий освобождается, супервизор берет первую задачу из буфера и назначает ее свободному рабочему, а на освободившееся место помещает новую подзадачу из базы данных Postgres. Если от бэкенда пришла новая, более приоритетная задача, супервизор может переписать весь или часть буфера. Подзадачи, которые находятся в буфере, имеют состояние ST-QUEUED, а подзадачи, которые еще ожидают попадания в него - INITIALIZED. Пример переноса подзадач в буфер и их назначения рабочим показан на Рис. 2.

Супервизор выполняет обработку команд и опрос рабочих в цикле. В каждом цикле супервизор выполняет четыре действия: собирает результаты рабочих, завершивших выполнение подзадач, обрабатывает команды бэкенда, назначает новые подзадачи освободившимся рабочим и заполняет буфер новыми подзадачами. По умолчанию, интервал между опросами составляет одну секунду, данный параметр может быть изменен. Для взаимодействия с рабочими используется REST API, которое будет описано далее, а для получения команд от бэкенда используется очередь Redis.

1.3. Рабочий

Сервис рабочий выполняет подзадачи, создавая и запуская docker-контейнеры из предоставленного пользователем docker-образа. Рабочий содержит два потока исполнения: в одном потоке работает HTTP сервер, в другом запускается

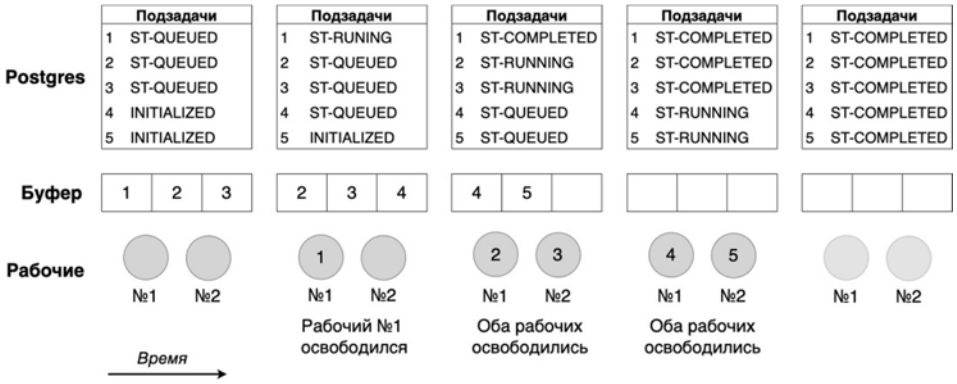

Рис.2. Пример состояния буфера в системе

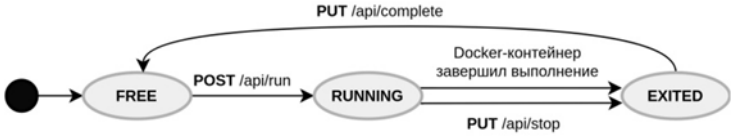

Рис.3. Диаграмма состояния рабочего

docker-контейнер с подзадачей. HTTP сервер предоставляет REST API для супервизора, чтобы супервизор мог отправить задачу на выполнение, проверить статус рабочего и завершить его работу. Сервис написан на Python, с помощью библиотеки Fast API.

На Рис. 3 показана диаграмма состояний рабочего. При старте сервис находится в состоянии FREE. После того, как приходит запрос от супервизора на выполнение, запускается docker-контейнер с подзадачей, и рабочий переходит в состояние RUNNING. В случае успешного выполнения рабочий переходит в состояние EXITED. После этого супервизор может получить результаты выполнения и удалить контейнер, в котором выполнялась подзадача, с помощью POST-запроса */api/complete*, после чего рабочий возвращается в состояние FREE и ждет поступления новых подзадач на выполнение.

1.4. Время выполнения задачи

Использование контейнеризации упрощает создание новых задач, делая систему независимой от того, какие технологии для выполнения расчетов нужны исследователю. Однако, любая форма виртуализации неизбежно ведет к появлению дополнительных накладных расходов. Также к их росту ведет асинхронное взаимодействие микросервисов.

Чтобы оценить накладные расходы, рассмотрим цикл выполнения подзадачи более подробно. На Рис. 4 показана временная диаграмма взаимодействия супервизора с рабочими сервисами. Для простоты рассмотрим случай, когда в системе только один рабочий. Как было отмечено выше, чтобы узнать состояние рабочего и проверить, доступен ли он для выполнения новых подзадач, супервизор реализует периодический опрос. В момент времени t_1 супервизор начинает опрашивать всех рабочих, посылая GET-запросы на получение статуса. После получения ответа, что сервис свободен (находится в состоянии EXITED) в момент времени t_3 супервизор обращается к Redis для того, чтобы взять из буфера задачу на выполнение, и отправляет ее рабочему (момент времени t_5), передавая POST-запрос /api/run. В момент получения запроса (t_6), рабочий запускает в отдельном процессе образ с подзадачей, выполняя команду **docker run**. Подзадача начинает свое выполнение в момент времени t_6. После ее запуска рабочий сообщает об этом супервизору (t_7). Если в системе есть еще рабочие, то супервизор продолжает опрос, завершая его в момент времени t_8.

В течение следующего опроса рабочий сервис все еще выполняет задачу, в ответ на запрос GET /api/status супервизор получает в ответе RUNNING. В следующую итерацию опроса (t_{12}) супервизор на запрос статуса получает ответ от

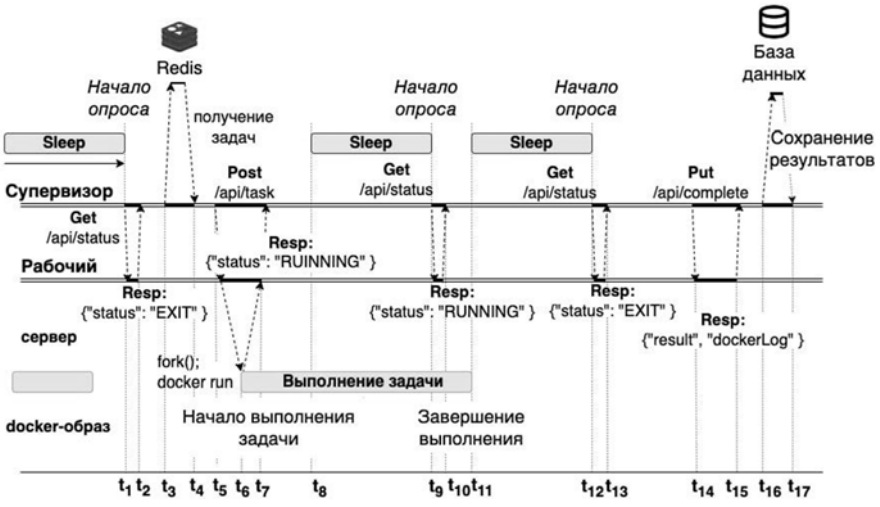

Рис. 4. Временная диаграмма выполнения подзадач в системе

рабочего, что он закончил выполнение и находится в состоянии EXITED. Супервизор, узнав, что рабочий закончил свое выполнение, отправляет PUT-запрос /api/complete на завершение выполнения контейнера. В интервале времени (t_{14},t_{15}) рабочий завершает выполнение docker-контейнера и удаляет его из системы, отправляет результат и лог выполнения супервизору. Супервизор получает ответ от рабочего и записывает результат выполнения задачи в базу данных. После этого супервизор может отправить новую задачу на выполнение рабочему.

2. Численные результаты

Для исследования производительности и эффективности применения системы мы провели ряд численных экспериментов. В начале раздела приведем пример задачи. Далее покажем, насколько использование контейнеризации увеличивает время выполнения задачи, а затем приведем результаты оценки влияния числа рабочих на длительность расчета при фиксированном числе процессоров. В ходе экспериментов оказалось, что выбор вычислительной платформы также оказывает существенное влияние на время выполнения задач. В завершение раздела покажем, как выбор стратегии приоритизации влияет на длительность полного расчета задачи и момента получения результатов первой подзадачи.

2.1. Пример задачи

Предположим, что пользователю нужно найти решение задачи “вычислить x в степени y” для различных точек (x,y). Будем считать, что в качестве языка программирования для реализации алгоритма был выбран Python. Пользователь составляет три файла: **input.json** с входным массивом значений параметров в формате JSON, **program.py** с программой, реализующий алгоритм расчета и **Dockerfile** с инструкциями по созданию docker-образа.

Файл **input.json** содержит в массиве перечисление всех точек, для которых нужно получить ответ. Для нашего примера **input.json** будет содержать следующие данные:

```
1 [
2   { “x”: 2, “y”: 0 },
3   { “x”: 2, “y”: 10 },
4   { “x”: 42, “y”: 3 }
5 ]
```

При написании программы и **Dockerfile**, где описываются инструкции для создания docker-образа, нужно учитывать следующие договоренности, которые позволяют единообразно передавать входные данные и получать результаты из контейнера: входные параметры для программы, работающей внутри контейнера, передаются через переменную окружения TASK_PARAMS; результат вычислений должен быть записан программой в формате JSON в файл /shared/results.json. После завершения docker-контейнера доступ к этому файлу сохранится, и рабочий, который выполнял контейнер, сможет передать результаты выполнения супервизору для сохранения в базе данных. Учитывая эти договоренности, программа в файле **program.py** будет выглядеть так:

```
import os
import json

params =json.loads(os.environ['TASK_PARAMS'])
with open('/shared/results.json', 'w') as f:
    result = {"result": params['x'] ** params['y']}
    print(json.dumps(result), file=f)
```

Наконец, пользователь должен составить **Dockerfile**, в котором содержатся инструкции по сборке контейнера. Предполагая, что файл program.py находится в той же директории, что и **Dockerfile**, последний содержит следующие инструкции:

```
FROM python:3
WORKDIR /app
COPY program.py .
CMD python program.py
```

В более сложных случаях **Dockerfile** может содержать инструкции по установке библиотек, компиляции и сборке программ, и прочие действия. После подготовки **Dockerfile** пользователь должен собрать его командой **docker build** и выложить в любой общедоступный репозиторий docker-образов командой **docker push** (например, hub.docker.com).

На вход системы пользователь должен будет передать путь к docker-образу в репозитории и файл входных параметров **input.json**. После выполнения задачи, пользователь сможет скачать результаты вычислений в формате JSON через веб-интерфейс. Каждая запись в файле результатов содержит значения входных параметров и результат выполнения. Так как подзадачи вычисляются асинхронно, порядок результатов в этом файле не определен. Для нашего примера файл результатов будет содержать следующие данные, с точностью до порядка записей:

```
[
 {"input": { "x": 2, "y": 0}, "output": {"result": 1}},
 {"input": { "x": 2, "y": 10}, "output": {"result": 1024}},
 {"input": { "x": 42, "y": 3}, "output": {"result": 74088}},
]
```

2.2. Оценка влияния контейнеризации на скорость вычислений

Для того, чтобы измерить накладные расходы при проведении расчетов в docker, мы написали тестовую задачу и сравнили время ее выполнения в операционной системе сервера без контейнеризации (Ubuntu 20.10) и внутри docker-образа. В качестве тестовой задачи мы использовали вычисление числа Фибоначчи рекурсивным способом, без применения динамического программирования. Этот алгоритм (который ни в коем случае не следует использовать для реального расчета чисел Фибоначчи!) удобен тем, что гарантировано загружает центральный процессор на достаточно продолжительное время, не требует переключений контекста исполнения в пространство ядра, и, таким образом, разные его запуски на одном и том же оборудовании требуют приблизительно одинакового времени. Исходный код программы на языке C++:

```
#include <chrono>
#include <iostream>
#include <iomanip>
#include <string>
#include <cstdlib>

using namespace std::chrono;

long fib(unsigned n) {
  if (n <= 1) return n;
  return fib(n-1) + fib(n-2);
}

int main(int argc, char **argv) {
  int n_iter = std::stoi(argv[1]);
  auto t_start = system_clock::now();
  long x{0};
  for (int i = 0; i < n_iter; i++) {
    x = fib(40);
  }
  auto t_end = system_clock::now();
  std::cout << std::fixed
     <<duration_cast<milliseconds>(t_end — t_start).count()
    << std::endl;
}
```

Функция *fib()* принимает на вход число Фибоначчи, которое нужно вычислить. В функции *main()* задается количество итераций вычисления числа. Отметим, что при компиляции не нужно указывать флаги оптимизации, так как в противном случае компилятор может удалить

Табл. 1. Время вычисления числа Фибоначчи рекурсивным способом на различных языках программирования

Язык программирования	Номер числа Фибоначчи	Время выполнения, мс
Python	30	482
Node JS	38	431
C++	40	600
Java	41	634

основной цикл в строках 18-20 из-за того, что результат вычисления (x) не используется в правых частях выражений.

В Табл. 1 приведено время выполнения программ, реализующих расчет чисел Фибоначчи, написанных на языках программирования Python, JavaScript, C++ и Java. Программа на JavaScript выполнялась с помощью Node JS, для Java использовался стандартный OpenJDK, для задачи C++ использовался компилятор Clang без оптимизаций. Мы подобрали номер числа и количество итераций таким образом, чтобы вычисление занимало порядка 0,5 секунд для каждой реализации. Стоит отметить, что код, написанный на языке Python, работал в десятки раз медленнее, чем код, написанный на других языках.

На Рис. 5 показана зависимость скорости вычисления числа Фибоначчи от количества итераций. Как можно видеть, время выполнения в docker-контейнере и в системе без контейнеризации отличаются. Наибольшая разница во времени выполнения наблюдается в реализации на языке Python, она достигает 15% при 50 итерациях. Для реализаций на C++, Java и Node JS отклонение для всех итераций не превосходит 5%. Таким образом, в зависимости от типа задачи и от используемого языка программирования время вычисления в docker-контейнере может быть гораздо больше времени вычисления на компьютере.

2.3. Зависимость времени выполнения задачи от числа рабочих

Супервизор проводит периодический опрос рабочих, в ходе которого назначает новые задачи и записывает результаты в базу данных. Длительность этого опроса зависит от числа рабочих и влияет на то, как долго рабочему придется ждать после завершения вычислений до начала обработки следующей подзадачи. Кроме того, из-за ограниченного числа vCPU на сервере между рабочими возникает конкуренция за процессорное время. Таким образом, число рабочих влияет на скорость вычисления.

Для оценки зависимости времени выполнения задач от числа рабочих мы провели серию экспериментов, в которых решали одну и ту же задачу вычисления числа Фибоначчи. Каждая задача состояла из 100 подзадач, для решения использовались облачные платформы:

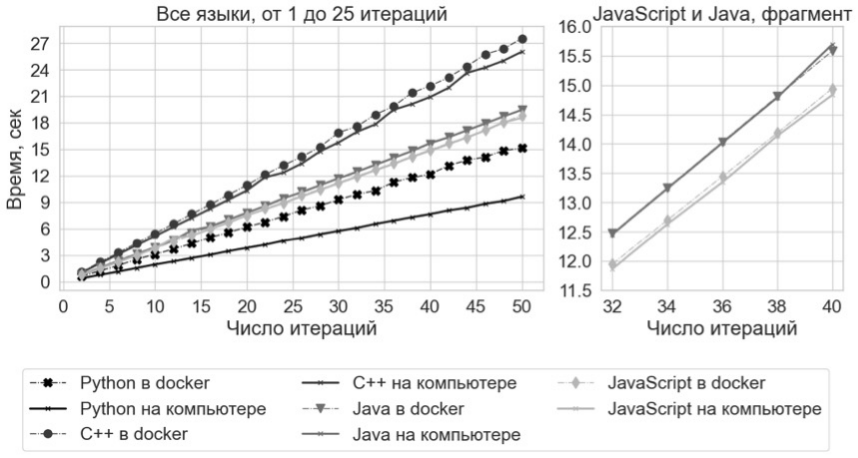

Рис.5. Сравнение времени выполнения задачи в докере и в системе без контейнеризации для различных языков программирования

- Yandex Cloud - процессор Intel Xeon Gold 6338, тактовая частота 2.00 Ггц;
- Selectel - процессор Intel Xeon Gold 6140, 2.3 Ггц;
- VK Cloud - Intel Xeon Gold 6230, 2.1 Ггц,

а также выделенная рабочая станция. Облачные серверы, использованные в экспериментах, имели по 8 виртуальных ядер. На рабочей станции был установлен процессор Intel Core i7-9700K, также имеющий 8 ядер и не поддерживающий hyper-threading. Кроме того, мы рассматривали два режима выделения ядер рабочим: динамическое, когда ядра назначаются процессам системным планировщиком, и фиксированное, когда ядра каждому вычислительному процессу назначались вручную.

Время выполнения каждой подзадачи составляло порядка 10 секунд, то есть на расчет всей задачи на одном процессоре потребовалось бы 10∗100=1000 секунд или 16 минут 40 секунд. Рабочие выполнялись в docker-контейнерах, для их запуска использовалась программа docker-compose. При запуске контейнеров с задачами рабочие обращались из отдельных процессов к docker-серверу хост-системы.

2.3.1. Теоретическая оценка длительности вычислений

Обозначим число процессоров как m и пусть n - число рабочих, p - число подзадач, а t_0 - длительность вычисления одной подзадачи одним рабочим на изначально свободном процессоре, то есть время вычисления подзадачи без учета времени ожидания освобождения процессора. Тогда теоретическая «оптимистичная» зависимость времени выполнения всей задачи от числа рабочих и числа процессоров в системе определяется как:

$$T(n,m,p) = \begin{cases} \frac{pt_0}{n}, n \le m \\ \frac{pt_0}{m}, n > m, \end{cases}$$

а среднее время расчета одной подзадачи t, учитывающее, что процессорное время может делиться между несколькими рабочими, определяется как: $\bar{t} = t_0 \max(1, \frac{n}{m})$.

2.3.2. Оценка накладных расходов. Влияние явной привязки рабочих к ядрам

Приведенные простые теоретические формулы не учитывают два фактора: время, расходуемое на переключение контекста выполняющихся процессов, если планировщик решает изменить выполняющуюся задачу (например, дать возможность выполниться какой-либо системной службе), а также время, расходуемое служебным процессом рабочего, который предоставляет API супервизору. На Рис. 6 приведена зависимость времени выполнения задачи на различных облачных платформах и рабочей станции в зависимости от количества рабочих.

Эксперимент был произведен при разных режимах запуска контейнера: без привязки рабочего к ядру, с привязкой рабочего к ядру. Кроме того, для рабочей станции был произведен эксперимент с привязкой к изолированным ядрам. Для привязки рабочего к ядру использовалась опция для контейнера cpuset_cpus, принимающая значение в диапазоне доступных ядер, для машины с 8 ядрами от 0 до 7 включительно. В эксперименте с изолированными ядрами все ядра, кроме двух, использовались только для вычислений, и они были запрещены для использования планировщиком с помощью параметра ядра Linux isolcpus=0-5, то есть на них гарантированно не выполнялись какие-либо системные процессы.

Не зарезервированные ядра 6 и 7 использовались как для вычислений рабочими, так и для всех системных служб и сервисных процессов рабочих. Результаты демонстрируют, что с точки зрения времени выполнения всей задачи, большого смысла в явной привязке рабочих к ядрам процессора нет; более того, явная привязка процессоров по рабочим дает небольшой выигрыш только в том случае, когда рабочих не больше числа зарезервированных ядер.

При выполнении задачи рабочим время тратится не только на само выполнение задачи, но и на ожидание загрузки, запуска и удаления контейнера, начала опроса супервизором, освобождения процессора от системного сервиса и прочие цели. Рис. 7 показывает различие между фактическим временем выполнения одной подзадачи и временем, которое требуется непосредственно для выполнения контейнера. Можно видеть, что при увеличении числа рабочих временные издержки растут. Это можно объяснить ростом времени, требуемого системе для переключения процессоров с вычислительных задач на служебные и системные процессы.

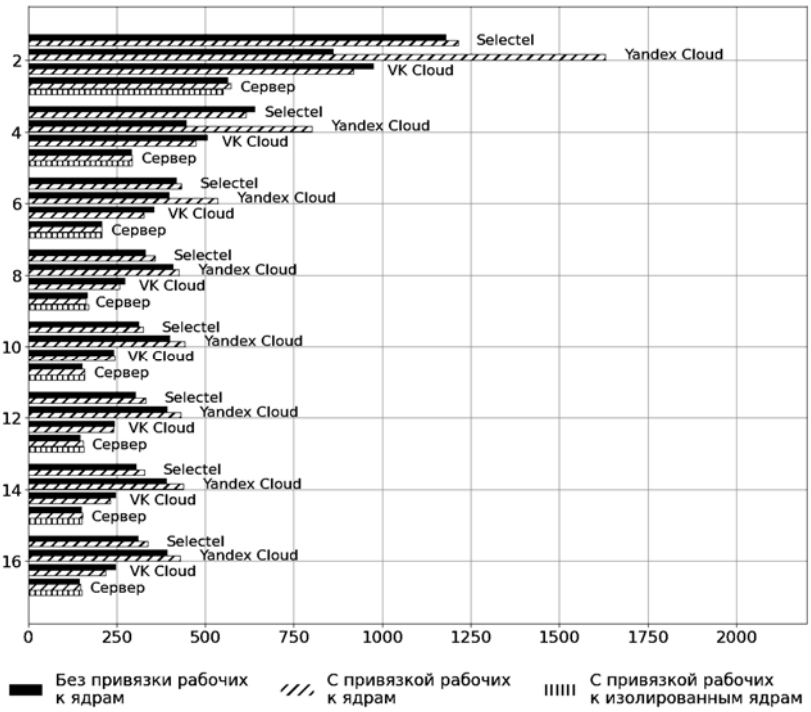

Рис.6. Время выполнения задачи при разных режимах выполнения: без привязки рабочих к ядрам, с привязкой рабочих к ядрам и с привязкой рабочих к выделенным ядрам

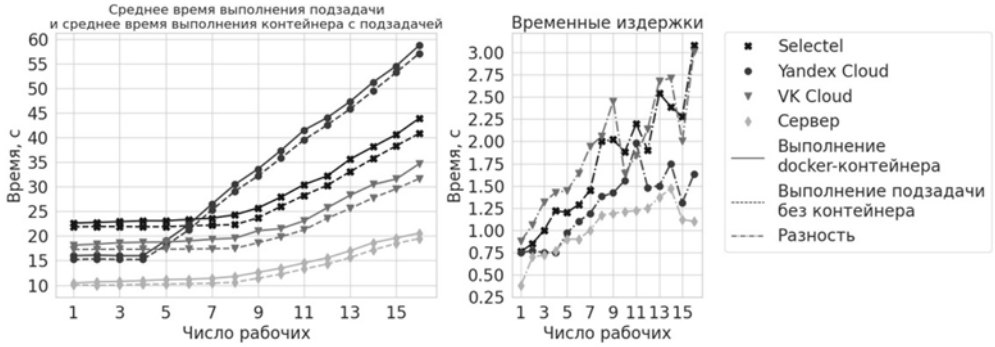

Рис.7. Среднее время выполнения подзадачи от числа рабочих в системе

Если рабочих больше, чем ядер, средняя длительность выполнения каждой подзадачи растет линейно, так как процессорное время делится между несколькими рабочими.

Интересный эффект при проведении эксперимента наблюдался при проведении эксперимента на серверах Yandex Cloud. На Рис. 8 показано среднее время выполнения подзадачи при разном числе рабочих в системе. Без привязки рабочих к ядрам время выполнения подзадачи начинает расти линейно, начиная с 4 рабочих, хотя количество виртуальных ядер в системе равно 8, и рост должен начаться, когда количество рабочих равно количеству ядер в системе.

При фиксации рабочих к ядрам в диапазоне от 1 до 8 рабочих при нечетном числе задействованных ядер наблюдается ускорение выполнения задачи. При четном числе рабочих наблюдается замедление выполнения подзадачи. Нам не удалось выяснить, почему происходит именно на этой облачной платформе. При фиксации рабочих к ядрам в диапазоне от 1 до 8 рабочих при нечетном числе задействованных ядер наблюдается ускорение выполнение задачи Возможно,

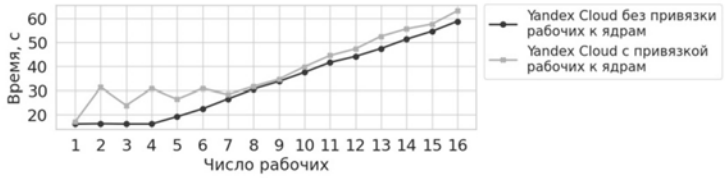

Рис. 8. Среднее время выполнения подзадачи на серверах Yandex Cloud от числа рабочих с привязкой и без привязки рабочих к ядрам

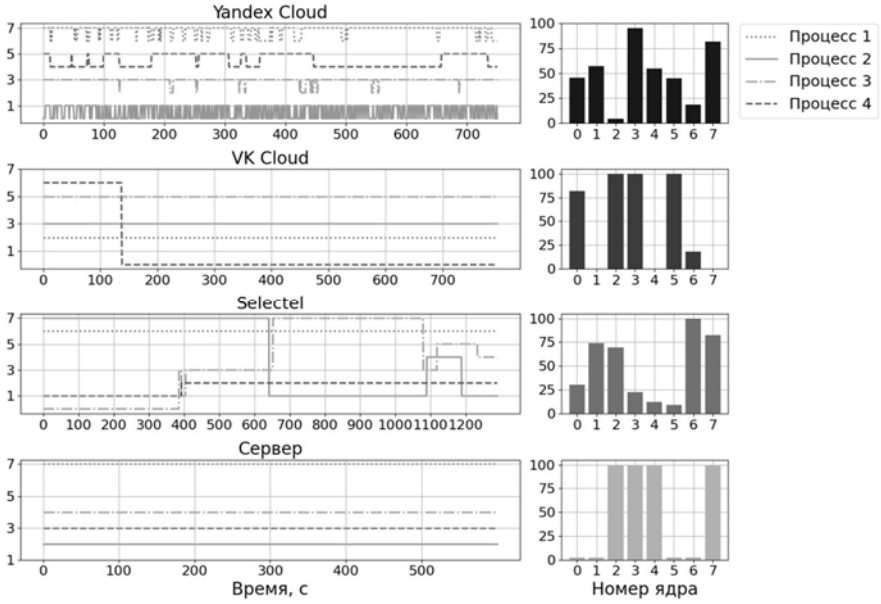

Рис. 9. Распределение процессов по ядрам при параллельном выполнении 4 задач на различных платформах

это связано с работой планировщика, а также двум виртуальным ядрам соответствует одно физическое, и при привязке рабочих к двум соседним виртуальным ядрам мощность одного физического делится пополам между рабочими.

2.3.3 Распределение процессорного времени между рабочими на разных платформах

В предыдущем подразделе было показано, что время выполнения задачи и подзадачи зависит от выбранной платформы для выполнения задачи. Время выполнения задачи зависит от количества ядер. Временные затраты на накладные расходы зависят от архитектуры виртуализации и работы гипервизора в системе.

Чтобы показать различие работы гипервизора на различных платформах мы провели эксперимент, в котором на платформе запускалось 4 параллельных процесса. В каждом процессе происходил численный расчет - вычисление числа Фибоначчи на C++. В процессе выполнения задач отслеживалось, какое ядро занимает каждый процесс. Для этого была написана простая программа, запускающая набор процессов, в каждом из которых выполняется расчет числа Фибоначчи, а в отдельном процессе параллельно, каждые 200 мс, считываются данные из файлов /proc/cpuinfo, /proc/stat и /proc/<pid>/stat. Исходный код программы доступен по адресу https://github.com/larioandr/cpustats. Как видно из Рис. 9, на некоторых платформах переключение между ядрами происходило достаточно часто, что накладывает дополнительные расходы на время выполнения задачи.

3.4. Приоритизация задач

Для исследования применения приоритетов мы провели численный эксперимент, в котором в систему приходили задачи двух видов: приоритетные (HP - High Priority) и неприоритетные (LP - Low Priority). В эксперименте изучалась эффективность применения разных стратегий приоритизации. Каждая задача содержит 100 подзадач. В эксперименте мы использовали

Табл. 2. Среднее время вычисления подзадач для разных режимов приоритизации

Режим приоритизации	Время выполнения с момента создания задачи до полного выполнения (сек)		Время до начала выполнения первой подзадачи (сек)	
	HP	LP	HP	LP
FIFO	526	589	446	501
PRIORITY FIFO	98	844	12	752
UNIFORM	1166	1159	8	12
PRIORITY UNIFORM	182	1542	5	84

рабочую станцию с Intel Core i7-9700K, также имеющий 8 ядер и не поддерживающий hyper-threading. В качестве подзадачи использовалось вычисление числа Фибоначчи, количество итераций и вычисляемое число подобраны так, чтобы вычисление занимало порядка 10 секунд (реализация на C++). Можно посчитать, что время выполнения 50 задач на 8 рабочих будет равно $\left[\frac{50}{8}\right] = 7$ - количество итераций запуска рабочих. Общее время равно произведению числа итераций запуска рабочих на время выполнения одной подзадачи: $7 * 10 = 70$ секунд. Приоритетные задачи поступают с интенсивностью одна в две минуты, а неприоритетные задачи - одна в полторы минуты.

В Табл. 2 приведены метрики для случая, когда каждая задача имеет постоянное время выполнения. Наиболее эффективной стратегией для выполнения приоритетных задач является PRIORITY FIFO, нахождение задачи в системе при таком режиме приоритизации минимально — 98 секунд. PRIORITY UNIFORM немного уступает PRIORITY FIFO во времени нахождения в системе задачи - 182 секунды, - но показывает лучшее время до начала выполнения первой подзадачи - 5 секунд. Если рассматривать случай, когда приоритеты не учитываются, то наиболее эффективной стратегией является FIFO. По сравнению с UNIFORM время нахождения в системе меньше и для приоритетных задач, и для неприоритетных. Большую часть времени в режиме FIFO задача находится в очереди на исполнение - т.е. время до начала выполнения первой подзадачи.

Заключение

В статье была предложена архитектура системы для организации параллельных вычислений с использованием технологии контейнеризации Docker и приоритизации задач. Мы описали архитектуру и взаимодействие сервисов внутри данной системы, предложили несколько стратегий для приоритизации: сортировка задач по времени поступления - FIFO, сортировка по приоритету и по времени поступления - PRIORITY FIFO, равномерное распределение задач по рабочим - UNIFORM и приоритетное распределение рабочих PRIORITY UNIFORM, когда все рабочие занимаются сначала приоритетными задачами. Для изучения эффективности каждой стратегии в работе проведен численный эксперимент. Наиболее эффективным для приоритизации задач оказался метод PRIORITY UNIFORM, при использовании которого время нахождения в системе приоритетных задач меньше в 10 раз, чем время нахождения неприоритетных задач. Численные эксперименты на разных облачных платформах показали, что эффективность существенно зависит от режима назначения виртуальных ядер физическим ядрам. На некоторых облаках наблюдались очень частые переносы исполняемых процессов между ядрами, что может быть следствием переназначения гипервизором виртуальных ядер. В системах, реализованных на основе OpenStack, результаты оказались наилучшими. При этом явная привязка рабочих к процессорам или изолирование ядер не показали существенного выигрыша с точки зрения времени расчета.

Литература

1. Vishnevsky V. и др. Performance evaluation of the priority multi-server system mmap/ph/m/n using machine learning methods // Mathematics. 2021. Т. 9. № 24.
2. Arora R., Redondo C., Joshua G. Scalable software infrastructure for integrating supercomputing with volunteer computing and cloud computing // Communications in Computer and Information Science. 2019. С. 105–119.
3. Nouman Durrani M., Shamsi J. A. Volunteer computing: Requirements, challenges, and solutions // J. Netw. Comput. Appl. 2014. Т. 39. № 1. С. 369–380.

4. Mengistu T. M., Che D. Survey and taxonomy of volunteer computing // ACM Comput. Surv. 2019. Т. 52. № 3.
5. Anderson D. P. BOINC: A Platform for Volunteer Computing // J. Grid Comput. 2020. Т. 18. № 1. С. 99–122.
6. Nikitina N. и др. Volunteer Computing Project SiDock@home for Virtual Drug Screening Against SARS-CoV-2 // IFIP Advances in Information and Communication Technology. , 2021. С. 23–34.
7. Barranco J. и др. LHC@Home: A BOINC-based volunteer computing infrastructure for physics studies at CERN // Open Eng. 2017. Т. 7. № 1. С. 379–393.
8. Brundo R., Nicola R. De. Blockchain-based decentralized cloud/fog solutions: Challenges, opportunities, and standards // IEEE Commun. Stand. Mag. 2018. Т. 2. № 3. С. 22–28.
9. Antelmi A. и др. A Volunteer Computing Architecture for Computational Workflows on Decentralized Web // IEEE Access. 2022. Т. 10. С. 98993–99010.
10. Agliamzanov R., Sit M., Demir I. Hydrology@Home: A distributed volunteer computing framework for hydrological research and applications // J. Hydroinformatics. 2020. Т. 22. № 2. С. 235–248.
11. Sukhoroslov O., Putilina E. Cloud Services for Automation of Scientific and Engineering Computations Science // Business. Soc. 2018. Т. 1. № 2. С. 6–9.
12. Abramson D., Giddy J., Kotler L. High performance parametric modeling with Nimrod/G: Killer application for the global grid? // Proceedings of the International Parallel Processing Symposium, IPPS. , 2000. С. 520–528.
13. Korenkov V. и др. The JINR distributed computing environment // EPJ Web of Conferences. , 2019. С. 03009.
14. Lamanna M. The LHC computing grid project at CERN // Nucl. Instruments Methods Phys. Res. Sect. A Accel. Spectrometers, Detect. Assoc. Equip. 2004. Т. 534. № 1–2. С. 1–6.

Соколов Александр Михайлович. Федеральное государственное бюджетное учреждение науки «Институт проблем управления им. В.А. Трапезникова Российской академии наук», Москва, Россия Научный сотрудник. Область научных интересов: исследование беспроводных широкополосных сетей, теория массового обслуживания, распределенные вычисления. Email: aleksandr.sokolov@phystech.edu

Ларионов Андрей Алексеевич. Федеральное государственное бюджетное учреждение науки «Институт проблем управления им. В.А. Трапезникова Российской академии наук», Москва, Россия. Старший научный сотрудник, к.т.н. Область научных интересов: математическое моделирование, телекоммуникационные сети, радиочастотная идентификация, распределенные вычисления. E-mail: larioandr@gmail.com

Вишневский Владимир Миронович. Федеральное государственное бюджетное учреждение науки «Институт проблем управления им. В.А. Трапезникова Российской академии наук», Москва, Россия. Заведующий лабораторией, д.т.н, профессор. Область научных интересов: исследование беспроводных сетей связи, теория массового обслуживания. E-mail: vishn@inbox.ru

Мухтаров Амир Амангельдыевич. Федеральное государственное бюджетное учреждение науки «Институт проблем управления им. В.А. Трапезникова Российской академии наук», Москва, Россия Старший научный сотрудник, к.т.н. Область научных интересов: математическое моделирование, телекоммуникационные сети, дискретная оптимизация, распределенные вычисления. E-mail: mukhtarov.amir.a@gmail.com.

Architecture of a Distributed Computing System with Tasks Containerization and Prioritization

A. M. Sokolov, A. A. Larionov, V. M. Vishnevsky, A. A. Mukhtarov

Institute of Control Sciences of the Russian Academy of Sciences, Moscow, Russia

Abstract. This article describes an architecture of a distributed system, which can speed up the process of obtaining results for such tasks. The system comprises a backend server, control service (supervisor), a set of worker nodes and a database. To abstract from particular languages and tools required for computational algorithms, these algorithms are executed in Docker containers. The system supports several strategies for tasks prioritization to operate efficiently under heavy load introduced by multiple users. To make use of the system, the user only needs to build a Docker image with an encapsulated algorithm, describe the input dataset in a JSON file and upload them via web interface. The system can be deployed in any public cloud. In this article, we provide a detailed description of the system architecture and numerical results obtained from computations on various clouds and local platforms. We show the influence of different prioritization strategies on the duration of computations under a moderate workload.

Keywords: parallel computing; container virtualization; cloud computing.

DOI 10.14357/20718632230401 **EDN** HSOYNI

References

1. Vishnevsky V. et al. Performance evaluation of the priority multi-server system mmap/ph/m/n using machine learning methods // Mathematics. 2021. Vol. 9, no. 24.
2. Arora R., Redondo C., Joshua G. Scalable software infrastructure for integrating supercomputing with volunteer computing and cloud computing // Communications in Computer and Information Science. 2019. Pp. 105–119.
3. Nouman Durrani M., Shamsi J. A. Volunteer computing: Requirements, challenges, and solutions // J. Netw. Comput. Appl. 2014. Vol. 39, no. 1. Pp. 369–380.
4. Mengistu T. M., Che D. Survey and taxonomy of volunteer computing // ACM Comput. Surv. 2019. Vol. 52, no 3.
5. Anderson D. P. BOINC: A Platform for Volunteer Computing // J. Grid Comput. 2020. Vol. 18, no 1. Pp. 99–122.
6. Nikitina N. et al. Volunteer Computing Project SiDock@home for Virtual Drug Screening Against SARS-CoV-2 // IFIP Advances in Information and Communication Technology. 2021. Pp. 23–34.
7. Barranco J. et al. LHC@Home: A BOINC-based volunteer computing infrastructure for physics studies at CERN // Open Eng. 2017. Vol. 7, no. 1. Pp. 379–393.
8. Brundo R. and Nicola R. De. Blockchain-based decentralized cloud/fog solutions: Challenges, opportunities, and standards // IEEE Commun. Stand. Mag. 2018. Vol. 2, no 3. Pp. 22–28.
9. Antelmi A. et al. A Volunteer Computing Architecture for Computational Workflows on Decentralized Web // IEEE Access. 2022. Vol. 10. Pp. 98993–99010.
10. Agliamzanov R., Sit M. and Demir I. Hydrology@Home: A distributed volunteer computing framework for hydrological research and applications // J. Hydroinformatics. 2020. Vol. 22, no 2. Pp. 235–248.
11. Sukhoroslov O. and Putilina E. Cloud Services for Automation of Scientific and Engineering Computations Science // Business. Soc. 2018. Vol. 1, no 2. Pp. 6–9.
12. Abramson D., Giddy J. and Kotler L. High performance parametric modeling with Nimrod/G: Killer application for the global grid? // Proceedings of the International Parallel Processing Symposium, IPPS. 2000. Pp. 520–528.
13. Korenkov V. et al. The JINR distributed computing environment // EPJ Web of Conferences. 2019. Pp 03009.
14. Lamanna M. The LHC computing grid project at CERN // Nucl. Instruments Methods Phys. Res. Sect. A Accel. Spectrometers, Detect. Assoc. Equip. 2004. Vol. 534, no 1–2. Pp. 1–6.

Sokolov Alexander M. V. A. Trapeznikov Institute of Control Sciences of Russian Academy of Sciences, 65 Profsoyuznaya street, Moscow 117997, Russia, e-mail: aleksandr.sokolov@phystech.edu

Larionov Andrey A. PhD, V. A. Trapeznikov Institute of Control Sciences of Russian Academy of Sciences, 65 Profsoyuznaya street, Moscow 117997, Russia, e-mail: larioandr@gmail.com.

Vishnevsky Vladimir M. Doctor of Technical Sciences, Professor, V. A. Trapeznikov Institute of Control Sciences of Russian Academy of Sciences, 65 Profsoyuznaya street, Moscow 117997, Russia, e-mail: vishn@inbox.ru

Mukhtarov Amir A. PhD, V. A. Trapeznikov Institute of Control Sciences of Russian Academy of Sciences, 65 Profsoyuznaya street, Moscow 117997, Russia, e-mail: mukhtarov.amir.a@gmail.com