Визуализация работы алгоритма планирования параллельных задач

П.А. Васёв [1]

[1] *Институт математики и механики им. Н. Н. Красовского Уральского отделения Российской академии наук, ул. Софьи Ковалевской, д. 16, Екатеринбург, 620108, Россия*

Аннотация

Работа посвящена вопросу визуализации работы алгоритма планирования параллельных задач. Планирование задач — ключевая часть разрабатываемой автором среды онлайн-визуализации и параллельного программирования. При программировании параллельной версии одной прикладной задачи возникло подозрение, что алгоритм планирования не вполне оптимально распределяет нагрузку между исполнителями. В связи с чем было решено визуализировать его работу, чтобы увидеть общую картину и возможные проблемные места работы алгоритма. Вид отображения работает в трёхмерном пространстве и визуализирует назначения задач точками. Координаты точек определяются логическим временем, порядковым номером ядра (исполнителя), и порядковым номером блока данных из декомпозиции задачи. Цвет точек задаётся типом задачи. Зависимости между задачами по данным показаны отрезками. Построенный вид отображения успешно справился с поставленной задачей, и алгоритм планирования был улучшен.

Ключевые слова

Визуализация, параллельные вычисления, визуализация программного обеспечения, визуализация алгоритмов, планирование задач.

3D visualization of HPC Tasks Scheduling Algorithm

P.A. Vasev [1]

[1] *N.N. Krasovskii Institute of Mathematics and Mechanics of the Ural Branch of the Russian Academy of Sciences, Sofia Kovalevskaya str., 16, Yekaterinburg, 620108, Russia*

Abstract

The paper is devoted to the issue of visualization of the algorithm for scheduling parallel tasks. Task planning is a key part of the online visualization and parallel programming environment developed by the author. When programming a parallel version of one application task, a suspicion arose that the scheduling algorithm does not optimally distribute the load between the performers. In this connection, it was decided to visualize its work in order to see the overall picture and possible problem areas of the algorithm. The display view works in three-dimensional space and visualizes the assignment of tasks with dots. The coordinates of the points are determined by the logical time, the sequence number of the core (performer), and the sequence number of the data block from the decomposition of the task. The color of the dots is set by the task type. Dependencies between data tasks are shown in segments. The constructed view of the display successfully coped with the task, and the planning algorithm was improved.

Keywords

Visualization, software visualization, high-performance computing, asynchronous many-task systems.

ГрафиКон 2023: 33-я Международная конференция по компьютерной графике и машинному зрению, 19-21 сентября 2023 г., Институт проблем управления им. В.А. Трапезникова Российской академии наук, г. Москва, Россия
EMAIL: vasev@imm.uran.ru (П.А. Васёв)
ORCID: 0000-0003-3854-0670 (П.А. Васёв)

1. Введение

Визуализация структуры и работы алгоритмов относится к области визуализации программного обеспечения [1]. Она, наряду с научной визуализацией и информационной визуализацией, включается в состав науки компьютерной визуализации [2].

Настоящая работа посвящена вопросу визуализации работы алгоритма планирования параллельных задач. Планирование задач является ключевой частью многих систем высокопроизводительных вычислений, в том числе и разрабатываемой автором среды онлайн-визуализации и параллельного программирования [3].

Среда построена вокруг идеи представления параллельного процесса вычисления в форме множества зависимых задач [4]. Зависимость задач означает, что входные аргументы для некоторой задачи определяются результатами вычисления других задач.

Список задач и зависимостей между ними определяется пользовательским алгоритмом, который может иметь как форму конечного вычисления, так и процесса. Задачи поступают на вход процессу планирования, который определяет, на каком вычислительном узле следует выполнять ту или иную задачу. Затем, по мере разрешения зависимостей, задачи выполняются. Естественно, что от качества используемого алгоритма планирования зависит эффективность всего параллельного вычисления.

Соответствие задачи узлу будем называть назначением. Набор таких назначений и зависимости между задачами будем называть *планом выполнения*. План выполнения, отметим, строится во времени, по мере поступления задач и проявления других аспектов.

Цель настоящей работы — предложить вид отображения (способ визуализации) плана выполнения. Эта цель обусловлена необходимостью понимания, насколько эффективно работает алгоритм планирования, как он себя проявляет в тех или иных условиях. Эта необходимость, в свою очередь, оказалась обусловлена одной практической задачей, решение которой вызвало подозрение в неоптимальности применяемого алгоритма планирования.

Структура работы следующая. В разделе 2 представлена среда параллельного программирования. В разделе 3 представлена прикладная задача и её параллельная версия. В разделе 4 представлен алгоритм планирования и требования к его визуализации. В разделе 5 представлен построенный вид отображения и выводы по полученной визуализации. В разделе 6 представлена модификация алгоритма планирования и его визуализация. В заключении представлен анализ полученного вида отображения и выводы по результатам его применения.

2. Среда параллельного программирования

Автор разрабатывает среду [3] для решения задач онлайн-визуализации и параллельного программирования. Онлайн-визуализация это визуализация суперкомпьютерных вычислений во время их выполнения. В отличие от визуализации после вычисления, онлайн-визуализация требует дополнительных инструментов для связи с работающей параллельной программой, чтобы получать от неё текущие данные и посылать управляющие сигналы [5,6].

В последнее время растёт объем вычисляемых данных, проводятся вычисления Exascale-масштаба. Сложилась ситуация, что процесс визуализации хода и результатов вычислений необходимо выполнять в параллельном режиме, также как и сами вычисления.

Заметим, что технически визуализация сама по себе является вычислением (конечно, с некоторыми особенностями). Поэтому была сформулирована следующая гипотеза: возможно предложить некий способ программирования, который позволит эффективно реализовывать алгоритмы визуализации, их связь с параллельными вычислительными программами, а также и сами сами эти программы, то есть произвольные вычислительные алгоритмы.

В качестве проверки данной гипотезы создаваемая среда предлагает следующий способ параллельного программирования.

Способ основан на понятии обещания. **Обещание**[2] (анг. *promise*, также используют термин *future*) это объект, который соответствует данным, которые будут вычислены когда-нибудь. Обещание можно создать в одном процессе, выполнить (анг. *resolve* и *fulfill*, то есть заполнить данными) в другом, а реагировать на выполнение — в третьих процессах.

Удобство обещаний заключается в том, что оперировать ими можно в любой момент времени, ещё до того как получены результаты вычислений этих обещаний. Это позволяет описывать параллельные алгоритмы обработки данных с помощью обычных последовательных кодов.

Предлагаемый способ параллельного программирования выражается следующей моделью.

Операция «добавить данные». Загружает данные в вычислительную среду.

add-data: data→ promise,

где *data* — загружаемые данные. Результат — объект обещания, которое находится в состоянии «исполнено». Данные *data* передаются в вычислительную среду.

Операция «добавить задачу». Добавляет заявку на вычисление задачи.

exec-request: action, args→ promise,

где *action* — действие, *args* — аргументы действия. Результат — объект обещания, который будет исполнен когда данная задача будет вычислена. Задача определяется действием и аргументами этого действия. Аргументы — это набор пар (имя,значение). Важно, что среди значений аргументов можно указывать обещания. Вычислительная среда выполнит указанную задачу 1) когда все обещания указанные в аргументах будут выполнены[3], и 2) по мере появления аппаратных возможностей для выполнения задачи.

Операция «получить данные». Выгружает данные из вычислительной среды.

get-data: promise → data,

где *promise* — объект обещания, data — данные, полученные в результате выполнения указанного обещания. Отметим, что эта операция возвращает не сами данные, а локальное обещание в терминах используемого языка программирования. Когда *promise* будет исполнен, исполнится и локальное обещание.

Также в модели реализованы дополнительные операции работы с обещаниями:

- **Создать обещание** *(create-promise)* — создаёт объект обещания.
- **Выполнить обещание** (*resolve-promise)* — выполняет указанный объект обещания, связывая его с указанными данными.
- **Операция «когда все»** *(when-all)* - создаёт объект обещания, которое будет исполнено, когда все обещания из заданного списка окажутся исполнены. Результатом выполнения такого обещания является список значений выполненных обещаний в порядке, соответствующему списку обещаний.
- **Операция «когда любое**» *(when-any)* - создаёт объект обещания, которое будет исполнено, когда будет исполнено любое одно обещание из заданного списка. Результатом выполнения обещания будет значение первого выполненного обещания.

Предполагается, что используя вышеуказанные операции возможно реализовать широкий класс вычислительных алгоритмов. Полагается, что вычислительный алгоритм описывает процесс вычисления в основном с помощью добавления задач в среду. При этом, если какие-то задачи независимы между собой по данным, то оказывается возможным их параллельное выполнение. Вычислительный алгоритм при этом **остаётся последовательным**.

Поступающие задачи передаются средой для исполнения в процессы, называемые исполнителями. При выборе исполнителя для очередной задачи, то есть при планировании задачи, среда оценивает состояние исполнителей и их загруженность. Более подробно архитектура и алгоритм планирования задач описаны в разделе 4.

[2] Понятие обещания и список операций с обещаниями описаны, например, в документации языка Javascript, раздел Использование промисов. Также статья Wikipedia: Futures and promises.

[3] В целом лучше, когда аргумент в форме обещания – единственный. Это соответствует математическим моделям вычислений, например, теории категорий. Такой подход применяется, например, в библиотеке C++ stdexec. Однако текущая реализация более лаконична, когда в аргументах задачи указываются все ожидаемые обещания (по сути это неявная операция «когда все»).

Дополнительно, для реализации интерактивных процессов среда предлагает взаимодействие с помощью передачи и приёма сообщений по модели издатель-подписчик (анг. publish-subscibe) с фильтрацией сообщений по темам (анг. topic-based publish/subscribe). Участники вычисления подписываются на темы сообщений. Другие участники посылают сообщения в те или иные темы. При посылке сообщения в тему оно доставляется всем тем участникам, которые подписались на эту тему.

Интересен вопрос, насколько применим представленный способ параллельного программирования. Предполагается, что данным способом можно решать как задачи параллельной визуализации (и в частности параллельного рендеринга), так и задачи более широкого класса.

3. Прикладная задача и её параллельный алгоритм

Для проверки применимости предложенного способа программирования было решено реализовать параллельную версию алгоритма задачи, описанной в [7]. Задача реализует метод аппроксимации динамики нелокального уравнения неразрывности нелинейной марковской цепью. Решение проводится в одномерном случае. Визуальный образ решения представлен на рисунке 1.

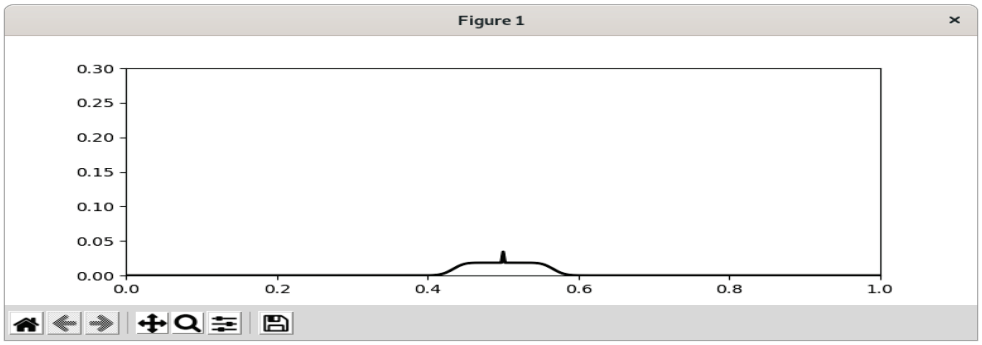

Рисунок 1 – Визуальный образ решения прикладной задачи при N=400, T=4000, см. [7]

Непараллельная версия алгоритма устроена следующим образом.

1. Обрабатываемые данные представлены массивом *mu* длины *N* с элементами типа *double64*. Начальное значение всех элементов *mu* — некоторая константа.
2. Запускается цикл из *T* итераций, где *T=10*N*.
3. На каждой итерации для каждого элемента *mu[i]* вычисляется «следующее» значение как функция вида *f(mu[i-1], mu[i], mu[i+1], expectation)*, где *mu[i-1], mu[i+1]* это «соседние» элементы *mu[i]*, а *expectation* – число *double64*.
4. По завершению каждой итерации вычисляется новое значение *expectation* как функция вида *g(mu)*, то есть функция от всего массива *mu*.

Параллельная версия алгоритма на основе представленного в разделе 2 способа параллельного программирования выглядит следующим образом.

1. Массив данных *mu* длины *N* разбивается на *P* частей одинакового размера (набор этих частей далее упоминается как «разбиение *P*»). Каждая часть загружается в среду посредством операции «добавить данные». В результате формируется набор обещаний *p_mu* длины *P,* и каждое такое обещание содержит привязанное значение *block* — массив элементов соответствующей части.
2. Запускается цикл из *T* итераций, где *T=10*N*.
3. На каждой итерации выполняется следующее:

3.1. Для каждого *p_mu[j]* запускается задача вычисления. Эта задача проводит вычисление элементов также, как и в непараллельном случае. На вход задача берет следующие аргументы:

- *block* – значение элементов части.
- *expectation* – параметр вычисления.

«Результат» вычисления записывается в обещание $p\_mu_{unsynced}[j]$ и он содержит:

- *block* – *н*овое значение элементов части.
- *left* – значения крайнего левого элемента части.
- *right* – значения крайнего правого элемента части.
- *average* – специальное значение, которое будет использовано в последствии при расчёте *expectation* данной итерации.

3.2. Для каждого $p\_mu_{unsynced}[j]$ запускается задача синхронизации граничных значений частей. Эта задача берёт на вход следующие аргументы:

- *outer_left = p_mu[j-1].right* - значение крайнего правого элемента блока слева,
- *outer_right = p_mu[j+1].left* - значение крайнего левого элемента блока справа,
- *block = p_mu[j].block* — свой блок

И вписывает значение *outer_left* в граничное левое значение блока данных, а *outer_right* в граничное правое. В результате запуска всех задач синхронизации формируется набор из *P* обещаний, который записывается в *p_mu* поверх старых значений.

3.3. Дополнительно, добавляется задача подсчёта *expectation*, которая берет на вход

- *array = when_all(* $p\_mu_{unsynced}$ *),* т.е. обещание, которое срабатывает при решении всех задач $p\_mu_{unsynced}$ текущей итерации, считывает значение *average* из всех *P* полученных результатов и по ним вычисляет итоговое значение *expectation*, которое будет использовано на следующей итерации.

Таким образом, всё вычисление выражается посредством *T*(P*2+1)* задач, связанных между собой по входам и выходам. Непараллельная версия алгоритма была написана автором алгоритма Ю. В. Авербухом на языке Python. Параллельная версия была реализована на основе непараллельной, также на языке Python.

Результаты измерения производительности работы алгоритмов показаны в таблице 1.

Таблица 1 – Производительность прикладной задачи

Способ распараллеливания	Миллионов операций в секунду
Непараллельная версия	~ **370**
С автоматическим распараллеливанием по потокам (16) и общей памятью numba.njit(parallel=True)	~ 986
Параллельная версия P=10 частей и W=5 исполнителях	~ **500**

Под операцией имеется ввиду расчёт одного значения в массиве *mu*. Все замеры проводились при параметре задачи $N=4*10^7$ на машине с процессором Ryzen 1700x. Все версии программ оптимизированы с помощью технологии numba с параметрами nopython, fastmath и библиотекой SVML.

Таким образом, полученная параллельная версия показала себя более скоростной, чем непараллельная версия. Но она уступила по производительности распараллеливанию на основе потоков. Возник вопрос, что могло послужить причиной такого отставания.

Среди гипотез были выдвинуты следующие:

При работе в многопоточном режиме с общей памятью **упрощается шаг синхронизации граничных значений.** На общей памяти перед началом вычисления блока можно считать граничные значения непосредственно из памяти, и затем использовать их как параметр вычисления. На распределённой памяти граничные значения необходимо передавать исполнителям явно, по сетевым или локальным протоколам межпроцессного взаимодействия, что влечёт соответствующую **задержку**.

Накладные расходы на назначение задач исполнителям. Эти расходы выражаются в **задержке**, которая проходит с момента фактической готовности очередной задачи, до момента начала выполнения этой задачи на каком-либо исполнителе. В непараллельной версии такая

задержка минимальна, ввиду того что «алгоритм формирования задачи» и алгоритм самой задачи размещены максимально близко. Конкретно, в непараллельной версии алгоритм формирования задачи это есть операция цикла по *T*, а алгоритм задачи — тело этого цикла. В многопоточной версии картина аналогична, хоть и несколько сложнее.

Такие задержки можно уменьшать. Например в обсуждаемой среде параллельного программирования была изменена архитектура. Вычисление готовности задач раньше работало до процесса планирования (на исполнители назначались только готовые задачи), а после изменения вычисление готовности стало работать на исполнителях (стали назначаться и неготовые задачи). Таким образом были исключены задержки от алгоритма планирования и от передачи данных к нему и от него. Также был применён более скоростной способ передачи информации между процессами среды: раньше сообщения передавались по протоколу HTTP с keep-alive, в новой версии они стали передаваться по протоколу TCP.

Неоптимальное планирование. После того, как задержки старта задач были по возможности минимизированы, возник вопрос — насколько качественно задачи назначаются исполнителям. Этот вопрос рассмотрен более подробно в следующем разделе.

4. Алгоритм планирования и требования к его визуализации

Алгоритм планирования это алгоритм, который выбирает, на каком исполнителе следует выполнять ту или иную задачу. В целом схема работы среды следующая:

1. Запуск пользовательской программы подразумевает запуск среды и набор процессов исполнителей (по выбору - на локальной машине, на вычислительном кластере).
2. Пользовательская программа связывается со средой и добавляет в неё задачи.
3. При добавлении задача передаётся алгоритму планирования.
4. Алгоритм планирования оценивает нагруженность и состояние исполнителей, и назначает задачи исполнителям. Назначение влечёт передачу информации о задаче выбранному исполнителю.
5. Каждый исполнитель имеет пополняемый список назначенных задач и отслеживает их готовность. Готовность задачи это готовность всех обещаний, указанных в её аргументах. По мере готовности задача начинает выполняться.
6. Выполнение задачи проводится в два этапа. Первый этап – подготовка аргументов. Аргументы могут быть константами, ссылками на обещания, или функциями. Во всех случаях кроме констант результаты подготовки аргументов заносятся в локальный кеш исполнителя. Это крайне существенная особенность среды, поскольку она позволяет многократно использовать повторяющиеся аргументы для разных задач, что существенно влияет на производительность. В частности, на этапе подготовки аргументов проводится загрузка данных, соответствующих обещаниям, в оперативную память исполнителя, возможно с других узлов вычислительной системы.
7. Второй этап — выполнение собственного алгоритма задачи. Результат работы задачи (блоки данных) остаётся в памяти исполнителя в кеше, а ссылка на него размещается в сопоставленном задаче обещании. Обещание переводится в состояние «выполнено».
8. По выполнению задач и как следствии их обещаний, становятся готовыми другие задачи, что и выясняется на шаге 5, и поток вычисления продолжается.

В представленной схеме важно отметить следующую деталь. Если результат работы задачи содержит большие по объёму данные, то в качестве данных исполняющих обещание, связанное с задачей, записываются не сами эти данные, а специального вида ссылки на них, а сами эти данные оставляются в оперативной памяти исполнителя. Когда какой-либо другой задаче понадобятся эти данные, то они либо а) повторно используются, если задача выполняется на этом же исполнителе, либо б) передаются в процесс исполнителя задачи, например копируются по сетевым протоколам.

Такой подход позволяет минимизировать количество передач данных между задачами, и допускает вовсе обойтись без таких передач. С другой стороны, он требует вести учёт доступной

оперативной памяти (узла или его устройств типа GPU, если данные размещаются в них) и проводить перенос данных по мере исчерпания памяти из процессов исполнителей в специальные процессы хранения.

Однако иногда такого переноса удаётся избежать за счёт следующей особенности. На практике зачастую нет необходимости проводить дублирование блоков данных. Например, в рассматриваемой прикладной задаче, описанной в предыдущем разделе, результаты расчёта очередной части *mu* можно записать «поверх» данных «старой версии» этой части.

Для учёта этой особенности в среду введена специальная функция преобразования аргумента задачи *reuse(promise)*, которая передаёт блоки данных оперативной памяти от обещания *promise* в сферу ответственности задачи, а само обещание при этом стирает (удаляет из среды). Задача может использовать эти блоки данных как входные, так и как выходные. В последнем случае задача может выдать свой результат на этих же самых блоках. Это и позволяет устранить дублирование – блоки оперативной памяти используются повторно.

При этом, если данные находились на другом исполнителе, то они также как и в случае «обычных» обещаний передаются в память исполнителя задачи, а в памяти исходного исполнителя — стираются.

В связи с вышеизложенными особенностями организации вычислительного процесса, требования к качеству работы алгоритма планирования задач становятся очевидными. Такой алгоритм должен проводить равномерную нагрузку задач на исполнителей, при этом стараться минимизировать количество передач данных между исполнителями, и при этом добиваться чтобы локальные кеши исполнителей использовались по возможности оптимально.

Алгоритм планирования.

Вход. На вход процессу алгоритма поступают задачи. Каждая задача описывается кортежем вида *(action,args,resources),* где *action* - идентификатор действия, *args* - аргументы, *resources* — требования к исполнителю по наличию устройств и (их) оперативной памяти.

Среди аргументов могут присутствовать такие, которые следует «вычислить» и оставить в локальном кеше. Например, загрузить блоки данных соответствующих обещаниям, или скомпилировать код функции, и т. п. Все подобные значения аргументов снабжаются глобальным уникальным идентификатором, который позволяет различать значения и размещать их в локальном кеше исполнителя. Список идентификаторов таких аргументов обозначен далее как $needs_{task}$.

Также на вход алгоритму с некоторой периодичностью поступает информация от каждого исполнителя *runner*: $queue\_size_{runner}$ - текущий размер списка назначенных и пока нерешенных заданий, $needs_{runner}$ - текущий состав локального кеша в форме списка идентификаторов.

Шаг алгоритма.

1. Назначение задач на исполнителей проводится поочерёдно и поштучно, по мере их поступления.
2. Для очередной задачи *task* для каждого исполнителя *runner* проводится оценка по формуле:

$$est = missing\_needs(task,runner) + queue\_size_{runner} * qcoef,$$

где $missing\_needs = | needs_{task} \setminus needs_{runner} |$ это количество аргументов задачи task, отсутствующих в кеше исполнителя *runner*, а вторая часть учитывает «нагруженность» этого исполнителя. Значение коэффициента $qcoef = 0.1$.

3. Задача назначается исполнителю с наименьшим значением *est*.
4. В локальную копию $needs_{runner}$ в памяти алгоритма вносятся все значения $needs_{task}$ и также идентификатор самой задачи, что означает что алгоритм в расчётах по последующим задачам будет считать эти значения уже находящимися в кеше (хотя фактически они попадут в кеш только когда задача будет выполнена).

Представленный алгоритм планирования, таким образом, не является алгоритмом об оптимальных назначениях, наподобие Венгерскому алгоритму. Вместо этого он пытается разместить поступающие задачи быстро, по мере их поступления, и при этом «разумно».

Было решено исследовать, как представленный алгоритм назначает задачи для описанной в разделе 3 прикладной задачи. Для исследования было решено применить визуализацию. **Требования к визуализации** были выдвинуты следующие:

1. Главная задача визуализации — дать понимание, есть ли ошибки или неоптимальность в поведении алгоритма планирования, и идеи по улучшению алгоритма.
2. Необходимо видеть распределение задач по исполнителям во времени.
3. При этом необходимо различать задачи по разбиениям (частям *mu*), чтобы понимать как именно распределяется нагрузка между исполнителями.
4. Необходимо отдельно видеть моменты копирования данных между исполнителями, так как перемещения данных оцениваются как наиболее затратный процесс, приводящий к задержкам в вычислениях.
5. Необходимо видеть зависимости задач от обещаний, в целях проверки корректности самой визуализации.

Конечно, это постановка была разработана не сразу, она начиналась с главной задачи, а последующие пункты уточнились со временем. Построенная визуализация (а точнее, вид отображения) представлена в следующем разделе.

5. Вид отображения

Исходя из требований, изложенных выше, построен следующий вид отображения, пример работы которого представлен на рисунке 2.

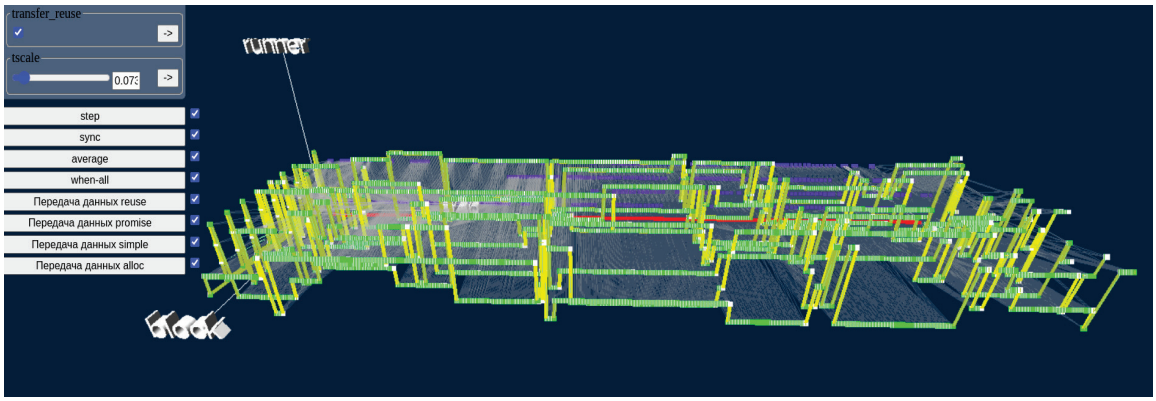

Рисунок 2 – Визуализация работы алгоритма планирования задач. Общий вид. Пользователь имеет возможность перемещаться в пространстве, погружаться внутрь и изучать детали, менять масштаб оси времени

Визуальный образ строится в трёхмерном пространстве.

- Ось X (слева направо на рис. 2) соответствует логическому времени алгоритма прикладной задачи, а именно номеру итерации.
- Ось Y (снизу вверх на рис. 2 - runner) соответствует порядковому номеру исполнителя.
- Ось Z (от дальнего к ближнему на рис. 2 - block) соответствует порядковому номеру части из разбиения P.
- Каждая задача показана точкой, цвет которой определяет тип задачи. Белые — счёт. Зелёные — синхронизация граничных значений. Синяя — расчёт *expectation*. Красная — примитив синхронизации «когда все», используемый при расчёте *expectation*.
- Зависимости между задачами показаны линиями (отрезками), идущими от одной задачи к другой. Поскольку задачи отображаются в логическом времени, то

зависимость задач интерпретируется в оси логического времени, задача позже во времени зависит от задачи раньше во времени.

- Тонкие линии показывают зависимости с передачей «лёгких» данных, например когда обещанию соответствуют данные в форме набора числовых констант.
- Яркие толстые линии показывают зависимости с передачей «тяжёлых» данных, например блоки памяти из разбиения *P*.
- Линии идущие от начала координат — это загрузка начальных данных.
- Вид отображения можно масштабировать по времени, для чего введён параметр масштаба, который меняется с помощью «слайдера» в графическом интерфейсе.

Вида отображения показывает важную характеристику по передачам данных. Каждая яркая толстая линия, переходящая по координате *Y*, означает передачу данных между исполнителями. Чем меньше таких линий, тем меньше передач данных происходит во время вычисления.

Визуальный анализ построенных планов выполнения задач показал следующее:

- Стало очевидно, что выбранную прикладную задачу можно решать вообще не передавая блоки данных между исполнителями. Такая передача оправдана, если какой-то исполнитель работает медленнее чем остальные, но эта информация на этапе планирования в текущей реализации алгоритма планирования недоступна.
- **Проблема 1**. Оказалось, что на начальной фазе все части разбиения планируются на одного исполнителя, затем происходит серия переброс на других, и лишь затем баланс находится и «стабилизируется», см. рисунок 3. Это хотя и логичное, но не вполне оптимальное поведение.
- **Проблема 2**. Оказалось, что во время работы существует множество необоснованных переходов между исполнителями. Так, видно что исполнитель решает серию итераций некоторого блока, а затем этот блок переназначается другому исполнителю. Что влечёт «разбалансировку» и массовый переход других блоков между исполнителями, рисунке 4.
- В целом вывод оказался следующий. Текущий алгоритм планирования не адекватен прикладной задаче, он влечёт очевидно излишние перемещения данных.

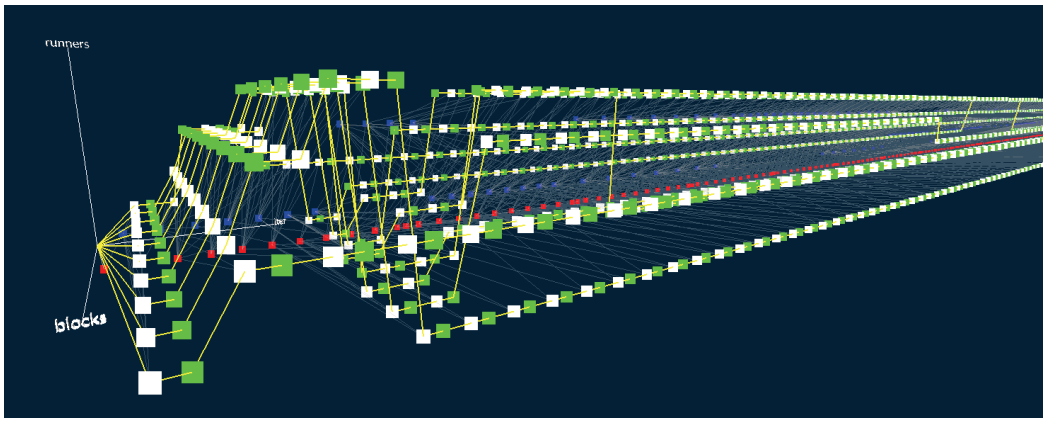

Рисунок 3 – Визуальный образ проблемы 1: показано начальное поведение алгоритма планирования на старте вычислений, видна странная перебалансировка — «перескок» всей нагрузки между исполнителями (вертикальная ось) вместо ожидаемой равномерной нагрузки

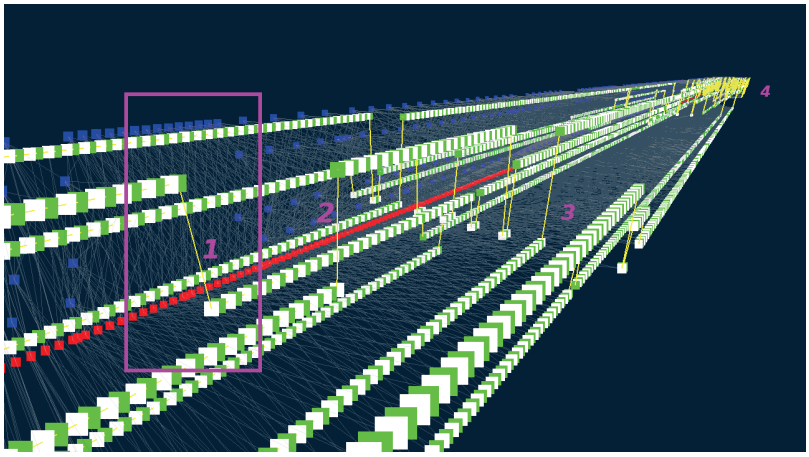

Рисунок 4 – Визуальный образ проблемы 2: «неспровоцированная» перебалансировка нагрузки. Фиолетовым прямоугольником (1) обозначена первая точка, когда нагрузка на расчёт блока была перемещена на другой исполнитель. Что повлекло последующую «разгрузку» этого исполнителя (2). Ещё через несколько итераций это привело к полной перебалансировке по всем исполнителям (3). На горизонте виден повтор подобного поведения алгоритма (4)

6. Модификация алгоритма планирования

Был проведён поиск причин обозначенных проблем, выявленных с помощью построенной визуализации. Среди предположений подтвердились следующие.

Причины проблемы 1. Как выяснилось, проблемное поведение вызвано тем, что алгоритм планирования при назначении задач наряду с наличием блоков данных учитывает также и компиляцию программных кодов. Программные коды задач подаются как специальный аргумент, и «стоимость» подготовки такого аргумента была такой же, как и «стоимость» передачи данных для задачи (см. формулу *missing_needs*). Поэтому при назначении первой задачи исполнитель, который получал её, получал и сверх-преимущество по отношению к другим исполнителям — ведь он «уже скомпилировал» код функции обработки данных. Это продолжалось до тех пор, пока размер очереди задач этого исполнителя не превышал определённого размера согласно коэффициенту *qcoef*, после чего преимущество нивелировалось и задачи начинали назначаться следующему исполнителю, который также начинал получать временное сверх-преимущество.

Решение. В принципе, поведение алгоритма «логично», и проблему можно было бы не решать. В качестве идеального решения можно принять введение весов при расчёте стоимости развёртывания аргументов задач. В качестве эксперимента было введён вес 0 для аргументов по компиляции кодов функций. Результат представлен на рисунке 5 слева.

Причины проблемы 2. Выяснилось, что причина кроется в скачкообразном изменении информации о размерах очередей задач исполнителей *queue_size*. Реализация системы такова, каждый исполнитель присылает обновление информации по своему состоянию периодически, несколько раз в секунду. При этом алгоритм планирования также моделирует размер этой очереди, увеличивая на 1 счётчик при каждом назначении задачи. Но никогда сам не уменьшает его, а ждёт информации от исполнителей. Получается, что при очередном получении информации «выясняется», что этот исполнитель уже решил ряд задач, его очередь существенно короче других. А более короткая очередь задач исполнителя, исходя из формулы (1), даёт преимущество этому исполнителю при назначении задач. Размеры этих скачков оказались таковы, что задачи «снимались» с исполнителя у которого развёрнуты блоки в оперативной памяти и передавались «более свободному» исполнителю. Конечно, через небольшое время приходило обновление и по другим исполнителям, но назначения уже были проведены. Фактически очередь исполнителя сокращалась, например, с 1614 до 1572 задач. Разница 42 при *qcoef=0.1* равная 4.2 и «перетягивала» задачи к этому исполнителю.

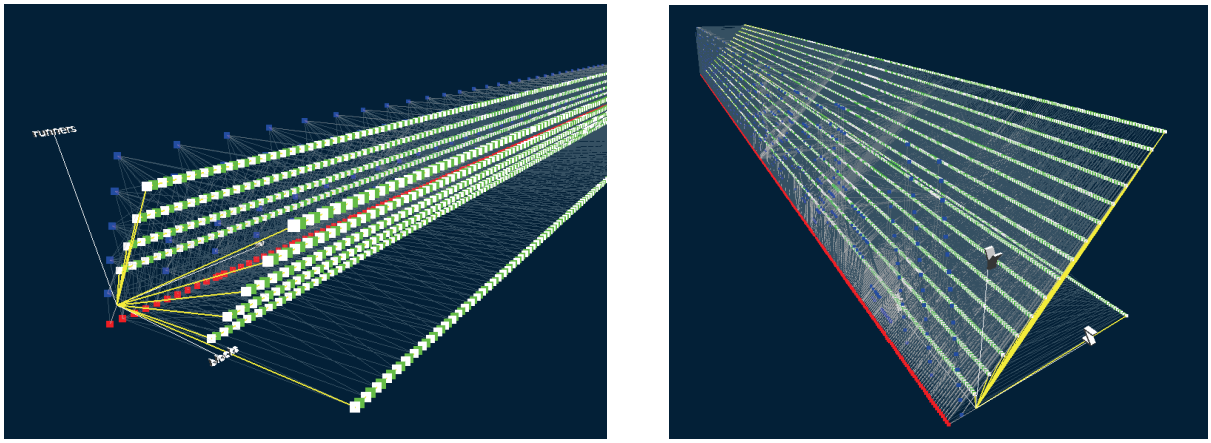

Рисунок 5 – Слева: визуальный образ решения проблемы 1**.** Задачи распределяются по исполнителям равномерно с самого начала. Однако на последующих итерациях всё ещё просматривается проблема 2 с избыточной перебалансировкой. **Справа:** визуальный образ решения проблемы 2**.** Показано 700 итераций при 16 частях и 16 исполнителях. Визуально наблюдается, что задачи по блокам распределяются по исполнителям равномерно на всём протяжении измерения. Назначение задач подсчёта *expectation* (точки синего цвета) неравномерно, но это не влияет на производительность вычисления, т. к. эти задачи не требуют передачи объёмных блоков данных

Решение. Среди вариантов решения был выбран следующий. Скачки в размерах очередей решено «сгладить» с помощью логарифма, формула (2):

$$est = missing\_needs(task,runner) + ln(\ queue\_size_{\ runner}\ )\ *\ qcoef, \quad (2)$$

где $missing\_needs = |\ needs_{\ task} \setminus needs_{\ runner} \setminus needs_{compile}|$ это количество аргументов задачи task, отсутствующих в кеше исполнителя *runner* и без учёта задач по компиляции кода $needs_{compile}$, а вторая часть учитывает «нагруженность» этого исполнителя и сглаживает скачки в обновлении информации в памяти алгоритма планирования. Параметр *qcoef=0.1*.

Результат решения проблемы представлен на рисунке 5 справа.

Выбранное решение проблемы 2 – спорное и «лобовое». Предварительно наблюдаются хорошие результаты, однако решение нуждается в дополнительной проверке и осмыслении. Например, при попытке убрать из формулы коэффициент *qcoef* в предположении, что логарифм сможет полностью сгладить скачки, привело к тому, что алгоритм планирования опять начал «троить», т.е. передавать ответственность над частями разбиения *P* от исполнителя к исполнителю. Также необходимо проверить, как эта формула поведёт себя не только в итеративных, а и в интерактивных задачах, например в задачах онлайн-визуализации.

Обновлённая информация по производительности следующая (таблица 2).

Таблица 2 – Производительность прикладной задачи

Способ распараллеливания	Миллионов операций в секунду
Однопоточная версия	~ 370
С автоматическим распараллеливанием по потокам (16) и общей памятью numba.njit(parallel=True)	~ 986
Параллельная версия P=10 частей и W=5 исполнителях	~ **500**
Параллельная версия с новым алгоритмом P=10 частей и W=5 исполнителях	~ **670**
P=10 частей и W=10 исполнителях	~ 740
P=16 частей и W=16 исполнителях	~ 630

Под операцией имеется ввиду расчёт одного значения в массиве *mu*. Все замеры проводились при параметре задачи $N=4*10^7$ на машине с процессором Ryzen 1700x. Величины *P=10* и *P=16* выбраны исходя из количества ядер процессора и удобства: в случае если *N* делится на *P* без остатка, то части разбиения *P* – одинаковы по размеру.

7. Заключение

Выводы по полученным результатам по алгоритму планирования следующие. Новая версия алгоритма (а точнее новая формула оценки исполнителей) показали ускорение с 500 до 670 миллионов операций в секунду. Этого удалось достичь за счёт ликвидации излишней передачи данных между исполнителями, вызванной неоптимальным назначением задач. Проблемы назначения были обнаружены визуально, с помощью представленного вида отображения.

В дальнейшем показатели были увеличены до 1388 миллионов операций в секунду (при P=4, W=4) за счёт переработки алгоритма прикладной программы:

- Задача синхронизации границ совмещена с задачей счёта (добавлена в начало).
- Расчёт *average* перенесён из отдельного цикла в цикл счёта *mu*.
- Задача расчёта *expectation* по значениям *average* частей совмещена с задачей счёта (добавлена в начало): таким образом подсчёт *expectation* дублируется в каждой части, но итоговые показатели эффективности оказались лучше, чем ожидание расчёта *expectation* как отдельной задачи.

При аналогичной оптимизации (расчёт *average* в цикле по *mu*) в версию с автоматическим распараллеливанием по потокам её показатели составили 1250 миллионов операций в секунду.

По представленному виду отображения выводы следующие. Его преимущества:

- Вид отображения справился со своей главной задачей — показать общую картину планирования задач и помочь визуально обнаружить существующие проблемы.
- Возможность масштабирования картины по логическому времени оказалась удобна, позволила изучать ситуацию как в целом, так и в подробностях.

Недостаток представленного вида отображения:

- Среди главных затратных элементов вычислений – транзакции синхронизации и передачи данных между исполнителями. Вид отображения успешно визуально выделяет передачи данных — линии при смене координаты *Y* хорошо заметны. Однако длина таких линий воспринимается как характеристика «дороговизны», «дистанции» передачи информации, а фактически она лишь равна разнице номеров исполнителей.

Перспективы развития вида отображения следующие:

- Вид отображения протестирован на практике на небольшом количестве «блоков данных» и исполнителей (до 100). Интерес представляет его проверка на большем количестве сущностей.
- Видеть одновременно как построенный план задач, так и фактическое его исполнение. Картина построенного плана показывает работу алгоритма планирования, а фактическое исполнение — как результаты работы этого алгоритма применяются в реальности, поскольку фактическое исполнение может существенно отличаться от планового ввиду неочевидных, но существенных особенностей. Пример визуализации фактического исполнения показан в видеоролике: https://youtu.be/XnV3l8hw8QE.

8. Благодарности

Автор благодарен коллегам из ИММ УрО РАН и УрФУ: Юрию Владимировичу Авербуху за предоставленную прикладную задачу; Сергею Владимировичу Поршневу, Илье Сергеевичу Стародубцеву, Михаилу Олеговичу Бахтереву, Сергею Владимировичу Шарфу, Дмитрию Валерьяновичу Манакову за участие и поддержку проекта.

9. Список источников

[1] Авербух В.Л., Авербух Н.В., Васёв П.А., Гвоздарев И.Л., Левчук Г.И., Мелкозёров Л.О., Визуализация программного обеспечения на базе средств виртуальной реальности геопространственных данных. Обзор и перспективы разработки // Известия Томского

политехнического университета. Инжиниринг георесурсов. 2020. Т. 331. No 1. 195–210. URL: https://www.cv.imm.uran.ru/e/3241749.

[2] Авербух В.Л., К теории компьютерной визуализации // Вычислительные технологии Т. 10, N 4, 2005 , стр 21-51. URL: https://www.cv.imm.uran.ru/e/3540 .

[3] Васёв П.А., Модель системы онлайн-визуализации // Параллельные вычислительные технологии (ПаВТ'2023) : Короткие статьи и описания плакатов, Санкт-Петербург, 28–30 марта 2023 года. – Челябинск: Издательский центр ЮУрГУ, 2023. – С. 236. – EDN IJALIW. URL: https://www.cv.imm.uran.ru/e/3241872.

[4] Котов В. Е., Проблемы развития параллельного программирования // Труды Всесоюзного симпозиума / Перспективы системного и теоретического программирования". Новосибирск, 1979. С. 58-72.

[5] Васёв П.А., Среда поддержки интерактивной визуализации для суперкомпьютерных вычислений // Вопросы атомной науки и техники. Серия: Математическое моделирование физических процессов. 2009. Выпуск 4. Стр. 67-77. URL: https://www.cv.imm.uran.ru/e/308951.

[6] Рябинин, К.В. Методы и средства разработки адаптивных мультиплатформенных систем визуализации научных экспериментов. Диссертация, Москва, 2015. URL: https://library.keldysh.ru/diss.asp?id=2015-ryabinin.

[7] Averboukh, Y. Lattice approximations of the first-order mean field type differential games. Nonlinear Differ. Equ. Appl. 28, 65 (2021). DOI: 10.1007/s00030-021-00727-2.