Метод схем для эффективного выполнения графов задач

П.А. Васёв[1] , С.В. Поршнев[1,2]

[1]Институт математики и механики им. Н.Н. Красовского
Уральского отделения Российской академии наук,
[2]Уральский федеральный университет им. Б.Н. Ельцина

Модель графов задач выглядит привлекательно для автоматизации программирования параллельных программ. Однако оказывается, что при ее прямом практическом применении возникают существенные накладные расходы. В настоящей работе предложено решение, как их избежать. Предлагается «метод схем». Пользователь описывает параллельную программу в виде схемы как набор взаимодействующих модулей. Каждый такой модуль определяет вычислительные процессы, которые запускаются на узлах суперкомпьютера. Взаимодействие модулей определяется связями между их портами ввода-вывода. На основе этих связей формируется следующий уровень связей – между вычислительными процессами. Взаимодействуя и реагируя на сообщения, процессы распределенно формируют и выполняют задачи графа задач. Предполагается, что данный метод будет удобен и эффективен.

Ключевые слова: суперкомпьютерные технологии, высокопроизводительные вычисления, параллельное программирование, граф задач, модели программирования.

1. Введение

Параллельное программирование – сложная задача. Чтобы справляться с ней, разрабатываются модели параллельных вычислений, которые структурируют и упрощают процесс мышления программиста. Например, разработаны модели:

- А-система Котова—Нариньяни [1] и тотальный параллелизм Котова [2];
- схема Карпа—Миллера [3, 4];
- непоследовательные процессы Петри [5] [1] и сети Кана [6];
- смешанные вычисления А.П. Ершова и идеи Л. Ломбарди о пошаговых вычислениях с неполной информацией [7];
- технология фрагментированного программирования [8, 31];
- машины абстрактных состояний Гуревича для параллельных алгоритмов [9];
- рекурсивные конвейеры [10] и модель bulk synchronous parallel [34];
- разнообразные модели Dataflow [4, 6], исчисления процессов (process calculi), например CSP Хоара [4], и другие.

Предназначение моделей вычислений состоит не только в том, чтобы человеку было удобно мыслить, разрабатывая алгоритмы. Но и в том, чтобы выраженные в этих моделях программы было возможно эффективно выполнять на машине. Под эффективностью здесь понимается соотношение затраченных аппаратных ресурсов и астрономического времени выполнения программы. Эффективность обеспечивается снижением накладных расходов, числа излишних операций, и т. п., вплоть до смены аппаратной базы. Если модель вычислений и удобна для человека, и эффективна для выполнения на машине, то такая модель «хорошая».

Меры удобства для человека бывают разные. Существуют чисто теоретические модели, которые хороши для суждений о свойствах алгоритмов, но неудобны в практическом применении (например, модель Тьюринга). Существуют модели и прикладные средства, такие как MPI, которые позволяют создавать универсальные и высокоэффективные приложения, но требуют высокой концентрации от программиста. Существуют и специализированные модели, которые

[1]Непоследовательные процессы Петри (Petri non-sequental processes, signal spaces) – модель, которая существенно отличается от широко известной модели сетей Петри.

крайне удобны и вычислительно эффективны, но в то же время не могут описать произвольные вычисления (например, такова модель map reduce).

Меры эффективности также бывают разные, и зависят от машин. Раньше машины использовали для ввода данных перфокарты и ленты, и важным аспектом было последовательное чтение данных, и программы должны были это учитывать. С появлением дисков с произвольным доступом эта потребность отпала. Другой пример: современные электронные процессоры и ускорители, которые диктуют, как следует формировать программы. Но в настоящее время разрабатываются и фотонные процессоры, и в случае их успеха уже эти технологии будут определять, какие операции следует использовать, как их следует группировать и т.п.

Для программирования параллельных программ определенной популярностью пользуется модель графа задач (task graph) [11–14]. В англоязычной литературе выделяют системы, работающие по этой модели, термином asynchronous many-task systems и task-based runtime systems, а модель программирования также называют sequental task flow.

Графы задач обладают рядом преимуществ, как в плане удобства мышления, так и эффективности выполнения, но имеют и существенные недостаток: при реализации «напрямую» они неэффективны на мелкозернистых задачах (т. е. на задачах с малым числом операций).

В настоящей работе предлагается некоторое улучшение для модели графа задач, которое устраняет обозначенный недостаток. Предполагается, что оно удобно для мышления и эффективно для выполнения в настоящее время, на текущем этапе развития. В будущем ситуация поменяется, и для уровней мышления программистов будущего и техники будущего найдутся другие, более подходящие модели и их улучшения.

В основе предлагаемого улучшения лежит переосмысление процесса динамического формирования графа задач, которое далее называется «метод схем»[2].

2. Модель графа задач

Согласно [13], модель графа задач можно определить следующим образом.

Граф задач – это ориентированный ациклический взвешенный граф $G = (V, E)$, в котором V – множество вершин, представляющих задачи, E – множество дуг, представляющих передачу данных между задачами. Также иногда дополнительно в E включают отношение предшествования между задачами[3]. **Задача** определяется как список инструкций (операторов), выполняемых последовательно на одном **исполнителе** (процессоре, ускорителе и т.п.).

Задача не может начать выполнение, не получив все входные данные. Следовательно, условием запуска задачи на выполнение является завершение выполнения всех задач-предшественников на графе задач. При выполнении этого условия задача готова к выполнению, которое произойдет далее в некоторый момент времени. Выполнение одних задач может происходить параллельно с другими задачами, и параллельно с передачей данных между задачами.

Понятие графа задач связано с понятием графа алгоритма [11]. Граф задач является образом графа алгоритма, т. е. граф алгоритма можно выражать в том или ином графе задач. Другими словами, граф алгоритма можно «упаковать» в тот или иной граф задач. Смысл такой упаковки – выполнение операций графа алгоритма группами, для снижения накладных расходов.

Назначение, на каком исполнителе в какой момент времени какую задачу следует выполнять – это отдельный процесс планирования (scheduling), поведение которого задается некоторым алгоритмом, который определяется или вычислительной средой, или пользователем.

Если алгоритм планирования определяется средой, то это может подразумевать динамическое планирование, исходя из текущей нагрузки на исполнители и других параметров. Это удобно для пользователя, но влечет накладные расходы. Если алгоритм определяется пользователем, это снимает расходы на динамическое планирование, но требует, чтобы пользователь сам определил, где следует выполнять задачи. Пользователь может определить это заранее,

[2]В настоящей работе слово «схема» используется в общем смысле, в контексте «схема параллельных вычислительных компонент», и не связано с понятием *схем программ*, введенным основоположником теории программирования А.А. Ляпуновым.

[3] На практике иногда применяются и более сложные методы описания взаимных зависимостей задач, например с указанием режима доступа к блокам данных (read-write-readwrite), см. например [18, 19].

статично, или определять динамически во время исполнения. Возможны также различные гибридные подходы, см. [16].

Преимущества модели графа задач:

1. Граф задач это удобная модель мышления для программиста. Параллельную программу в этой модели можно определять последовательным алгоритмом. Последовательные алгоритмы понятны и удобны для человека.
2. Возможность динамического и автоматического (без участия пользователя) балансирования нагрузки на множестве исполнителей.

Главное преимущество модели графа задач заключается в ее удобстве для мышления программиста. Используя метафору графа задач, возможно описание параллельного вычисления, причем формировать это описание возможно последовательным алгоритмом.

Например, такой алгоритм может быть устроен следующим образом (более подробное описание дано в [15, 17]). Пусть есть вычислительная среда, выполняющая задачи в модели графа задач. Пусть среда предоставляет операцию добавления задач. В результате выполнения этой операции немедленно формируются указатели на будущие результаты выполнения этих задач так называемые обещания (promise и future). Эти обещания можно использовать как аргументы при вызове других операций добавления задач. Заметим, что операции добавления задач в среду выглядят как обычные вызовы функций, и результаты тоже выглядят как обычные результаты срабатывания функций. Таким образом, алгоритм формирования графа задач можно записать в последовательной форме, и никаких дополнительных ухищрений не потребуется. Получается, что программист работает с привычной и удобной метафорой последовательного программирования. При этом фактическое выполнение задач может происходить параллельно, исходя из решений, принимаемых в процессе планирования (назначения задач исполнителям).

Модель графов задач настолько удобна для мышления, что она позволяет мыслить о параллельных вычислениях, выраженных и в других моделях. Например, в модели графов задач удобно представлять итеративные программы (то есть программы, проводящие вычисления над каким-то блоком данных в цикле), написанные на MPI. Так, тело внешнего цикла каждого из MPI-процессов в модели графов задач есть запуск списка задач, где каждая итерация представляется отдельной задачей. Блокирующее получение MPI-сообщений в модели графов задач выражается как запуск задач-продолжений (англ. continutations), в качестве аргументов которым указываются обещания результатов других задач, возможно с других исполнителей. Если в MPI при блокирующем получении сообщений с внешними результатами ожидание выражается явно, то в графе задач достаточно предъявить объект обещания этих результатов[4,5].

Среди систем на основе модели графа задач: SYCL task graph, CUDA Graphs, OpenTS, Template Task Graphs, PaRSEC, HPX, Charm++, Uintah, Kokkos, Legion, C++ Sender Library, LuNA, Dask, Flyte, TaskTorrent, OpenMP tasks, и другие. Они различаются между собой сферой применения, наличием своего языка программирования, и другими нюансами.

Также отметим систему Parallel Programming Kit разработки одного из авторов настоящей работы, которая описана в [15–17, 33] и опубликована на сайте github.com/pavelvasev/ppk.

3. Проблемы реализации модели графа задач

При реализации модели графов задач могут проявляться (и проявляются) разнообразные проблемы, среди которых в настоящей работе предлагается сфокусироваться на следующих:

[4]Отметим, что чтобы получить обещание результата вычислений с другого исполнителя (например узла суперкомпьютера), нет необходимости общаться с ним. Достаточно договориться о порядке именования такого обещания.

[5]Модель графа задач удобна для интерпретации и обычных ситуаций. Например, иногда программных интерфейсах присутствуют блокирующие вызовы, такие как работа с файлами в языке Си – fopen, fwrite, fread. При необходимости одновременного выполнения операций в таких интерфейсах пользователи вынуждены создавать нити, применять мьютексы для обмена данными между нитями, и т.п. В модели графов задач происходящее выглядит следующим образом. Блокирующий вызов операции это есть запуск двух задач: собственно операции и алгоритма продолжения. При необходимости одновременного выполнения операций создается столько задач, сколько необходимо. «Нити» при этом не нужны.

П1. Накладные расходы, связанные с передачей задач по сети, в первую очередь задержки по времени.

П2. Необходимость прямого доступа к содержимому блоков данных результатов задач из других задач без копирования памяти.

П3. Необходимость систематизации и минимизации количества аллокаций блоков данных.

П4. Необходимость предварительной инициализации (структур данных, библиотек и т.п.).

Проблема **П1** это накладные расходы, которые появляются в связи с обслуживанием задач. Если задачи формируются в некотором процессе, а выполняются в других процессах, то эту задачу надо добавить в среду, запомнить в некоторой форме, отследить момент готовности этой задачи к исполнению (если в аргументах есть обещания с результатами других задач), провести планирование и выбрать исполнителя, передать по сети процессу исполнителя, и т.п. Все эти моменты, каждый ничтожный сам в себе, в сумме накапливаются. С ростом числа задач они начинают занимать существенную часть как оперативной памяти, так и вносить существенные задержки по времени.

Всякое более менее серьезное вычисление может содержать миллиарды задач и более. Каждая задача из этого миллиарда проходит по разным узлам вычислительной среды, передается по сети и т. д. Если каждая задача содержит в себе относительно небольшое количество операций исполнителя (например 10^6 арифметических действий), а на обслуживание этой задачи уходит сопоставимое количество операций, то использование модели графа задач становится неэффективным и невыгодным.

Проблема **П2** поясняется следующим. Пусть некая задача T произвела блок данных[6]. Модель графов задач подразумевает, что этот блок данных передается на вход некоторым другим задачам. Возникает вопрос, как именно он передается: посредством копирования или без него. Если задача подразумевается к исполнению на другом узле, копирования и передачи данных по сети не избежать. Если задача подразумевается к исполнению на этом же узле, где произведен блок данных, то копирование можно делать, а можно не делать. Важный аспект заключается в том, что с учетом скоростей работы современных процессоров копирование данных в оперативной памяти это очень затратная операция, сопоставимая с самими вычислениями. Если вычисление занимает лишь несколько тактов на элемент блока данных, то получается что копирование блока данных по временным затратам сопоставимо с самими вычислениями. Поэтому существенно и важно, чтобы блоки данных в оперативной памяти при передаче между задачами не копировались.

Проблема **П3** модели графов задач связана с тем, что на практике не получается просто взять и начать решать задачи. Каждой задаче необходима оперативная память для размещения результатов ее работы. Можно получать эту память обычным способом, путем ее запроса (аллокации) у операционной системы (ОС) исполнителя. При этом:

1. Операция аллокации памяти это «дорогая» операция. Она настолько дорогая, что ее чрезмерное применение может в разы снижать производительность вычислений [32].

2. Аллоцированную память необходимо освобождать, как только она перестает быть нужной. Однако не вполне ясно, когда именно можно освобождать ту или иную память в условиях среды, когда результаты задач используются другими задачами, а граф задач строится динамически внешним алгоритмом.

Проблема **П4** подразумевает, что иногда для решения задачи необходимо провести дорогостоящую предварительную подготовку. Например, инициализировать ту или иную библиотеку, или подготовить особые структуры данных в памяти. Иногда эти операции можно было бы не повторять для решения однотипных задач. Но модель графа задач не учитывает этой особенности.

4. Обзор литературы

Принципиально проблемы **П2** и **П3** рассмотрены еще в 60х годах, см. [1, 3], и на современном уровне решение развивается например в [20, 21, 31]. Общая идея заключается в примене-

[6]Под блоком данных понимается блок оперативной памяти, в котором хранятся «тяжелые» данные – вычислительные сетки и т.п.

нии той или иной модели доступа к оперативной памяти. Вместо аллокации памяти, вводится пул буферов. Память не аллоцируется, а используется тот или иной буфер. В один момент времени буфер передается в управление одной задаче, в другой момент времени – другой задаче.

Например, в работе [21] описывается концепция на основе частичного порядка операций. Это определяет, в какой момент времени каким задачам следует выполняться. Это определяет и время, когда та или иная задача может иметь доступ к блокам памяти. Операции подразумеваются над общей памятью. Таким образом разные задачи имеют возможность доступа к одним и тем же блокам памяти, а конфликты устраняются заранее программистом, который формирует частичный порядок тем или иным алгоритмом.

Проблема **П1** решается на современном уровне по-разному. Общая идея заключается в том, чтобы так или иначе оптимизировать хранение и передачу данных о задачах на исполнители, а в идеале – и вовсе отказаться от такой передачи.

В работах [19] и [22] используется понятие *Parametrized Task Graph*, которое заключается в следующем. Граф задач представляется в форме функции, которая по аргументу (параметру) возвращает очередной участок графа. На исполнители передается не сам граф (и таким образом не сами задачи и зависимости между ними), а эта функция. Исполнители вызывают ее по мере продвижения по графу, и получают таким образом информацию о новых задачах, их выходах, и входах. Таким образом, отсутствует необходимость передачи задач по сети, а точнее задачи и граф в целом передаются в более компактной форме. Минус этого подхода – статичность функции порождения графа (т. е. неизменность по ходу вычисления) и высокая сложность ее описания для программиста, что отмечают и авторы.

В работе [18] предлагается другой подход. В нем граф порождается с помощью обычных операций добавления задач в среду, но на алгоритм добавляющий задачи накладывается ограничение: 1) алгоритм порождения графа должен быть описан в форме, которую можно передать по сети, и 2) при добавлении задачи сразу должен указываться исполнитель для нее, т. е. планирование назначения задач исполнителем ограничено моментом постановки задачи. Далее среда – разносит указанный алгоритм по исполнителям, и запускает его на каждом исполнителе. При добавлении очередной задачи исполнитель берет задачу только в случае, если она предназначена для него, а в противном случае игнорирует эту задачу.

Таким образом, как и в подходе [19], вместо графа на исполнители передается функция порождения графа. Отличие заключается в семантике и форме этой функции. В [19] это функция описывает процесс порождения граф задач и она параметризована идентификаторами, соответствующими некоторой нумерации узлов графа. А в подходе [18] эта функция существенно проще – это обычный алгоритм порождения графа посредством вызова операции добавления задач. Некоторый минус этого подхода – происходит дублирование вычисления функции порождения графа на всех исполнителях.

Ранее авторами настоящей работы был предложен метод итераций, описанный в работе [23]. Пользователь формирует подграф графа задач такой, который составляет одну итерацию вычислительного процесса. Затем пользователь имеет возможность выполнить операцию запуска этого подграфа указанное количество раз на указанном множестве исполнителей. При этом среда обеспечивают корректную связь выходных каналов задач подграфа с входными каналами задач подграфа следующей итерации. Более того, среда обеспечивает автоматизацию формирования структуры подграфа задач в оперативной памяти, перехватывая вызовы операции добавления задач в граф задач. Однако эта возможность представляется излишней, и удобнее не обеспечивать такую автоматизацию, а предлагать явные средства по формированию подграфа итерации. Кроме того, этот подход применим только для описания итеративных вычислений.

В технологии C++ Sender Library предлагается строить граф задач с помощью двух стадий: построение шаблона и запуск шаблона. Это позволяет оптимизировать передачу задач, потому что при повторном использовании одного и того же шаблона можно передавать по сети не описание задач, а идентификатор шаблона. Более того, есть операция запуска цикла *repeat_n*, которая означает запуск заданного подграфа несколько раз. Научные вычисления зачастую итеративные, и поэтому операция запуска цикла способна решить проблему накладных расходов на передачу задач и на их запуск. Аналогичный подход используется и в CUDA Graphs.

Проблема **П4** решается в проектах как правило путем введения в вычислительную модель акторов (англ. actor), например см. *chare* в Charm++. Также используется термин компонента (HPX, Dask). Задача в этом подходе решается не вычислительной средой, но конкретным экземпляром актора. Он может провести предварительную инициализацию и использовать ее результаты для решения поступающих экземпляру задач.

Все перечисленные способы в принципе решают обозначенную проблему накладных расходов графов задач. Но вопрос – какой ценой. Одни методы требуют взамен усложнения способа описания графа задач. Другие – проще в работе, но ориентированы исключительно на итеративные расчеты и фиксирует структуру одной итерации. Кроме того, методы не всегда подразумевают взаимодействия задач итераций с внешним миром.

5. Метод схем

Для решения проблем **П1-П4** предлагается следующий метод.

Используем метафору блок-схемы. С помощью записи, похожей на блок-схему, будем определять параллельный вычислительный процесс. А именно, схема – это структура, которая состоит из модулей, их портов, и связей между портами. Модули определяют вычислительные процессы, которые выполняются на исполнителях и взаимодействуют посредством связей выходных и входных каналов. А эти вычислительные процессы, в свою очередь, будут порождать и выполнять задачи из модели графа задач. Таким образом, схема определяет распределенный алгоритм динамического построения графа задач и алгоритм планирования его выполнения. Обратимся к деталям.

Исполнитель – однопоточный процесс уровня ОС. Исполнители запускаются на узлах суперкомпьютера. Исполнители можно использовать разные, исходя из оборудования и языковой среды. Например если на узле 64 ядра и 2 GPU, то можно запустить 32 исполнителя для двоичных кодов, 32 для языка Python, и 2 для GPU. Задача исполнителя – быть местом выполнения вычислительных процессов. Один исполнитель может выполнять сразу множество вычислительных процессов, благодаря их структуре, описанной далее.

Вычислительный процесс – это логический процесс, задача которого проводить конкретные вычислительные операции над входными данными и выдавать результаты.

Введем структуру вычислительного процесса. Процесс ожидает данных в форме сообщений из входных каналов (см. далее). По получению данных процесс проводит *вычисление*, и записывает результаты в выходные каналы, и возвращается в ожидание сообщений из входных каналов[7]. Вычисление непрерываемое, неблокирующееся и не длительное (подходящее значение длительности определяется экспериментально).

Благодаря такой структуре становится возможным размещение не одного, а множества вычислительных процессов на одном исполнителе. Логически, эти процессы выполняются все одновременно. Технически же они выполняются поочередно, выполняют свои вычисления по мере поступления сообщений.

Канал – это коммуникационный примитив. Поддерживаются операции: записать в канал, прочитать из канала. В канал можно записывать значения (т. е. сообщения) произвольной структуры, чтение и запись атомарны. Каналы бывают двух типов: 1) локальные по отношению к процессам, 2) глобальные, с общим на всю схему пространством имен каналов. Используя эти имена, любой процесс может взаимодействовать с любым каналом[8].

Связь между каналами – это коммуникационный процесс, который обеспечивает передачу сообщений между двумя каналами, а именно от канала-источника к каналу-приемнику. Один

[7] Другими словами, процесс ожидает сообщения, и реагирует на их поступление, выполняя алгоритм *реакции* (в смысле reaction [24]). Этот алгоритм и реализует «вычисление». Вычислительные процессы в настоящей работе похожи на процессы Хоара.

[8] Также прорабатывается вариант, в котором вводится два типа каналов – локальные по отношению к процессам и глобальные в общем пространстве имен. В этом варианте процессы взаимодействуют только со своими локальными каналами, а общее взаимодействие определяется связями этих локальных каналов с глобальными каналами.

вычислительный процесс может записывать сообщения в некоторый канал, а другой процесс читать сообщения из некоторого другого канала. Если эти каналы соединить связью, будет происходить передача сообщений от первого канала ко второму и таким образом – от первого процесса ко второму. Ограничений на количество связей, с которыми взаимодействует канал, не вводится.

Перечисленные выше сущности принадлежат «низкому» уровню, и не присутствуют в схемах напрямую. Схемы предлагается составлять из следующих сущностей:

Модуль – это виртуальный логический процесс преобразования данных. Задача модуля – определить, на каких исполнителях какие вычислительные процессы следует запустить. Список доступных модулю исполнителей сообщается модулю внешним образом, например в момент добавления модуля в схему. Кроме того, каждому модулю соответствует набор портов ввода и набор портов вывода.

Порт – это виртуальный, логический коммуникационный примитив, предназначенный для взаимодействия модулей. Порты разделяются на входные и выходные, и «прикрепляются» к модулям. Взаимодействие модулей осуществляется посредством связей между их портами. Технически порт – это множество каналов («пучок каналов» – выражение М.О. Бахтерева). Более того, это множество определенной структуры. Эта структура может быть списком, двумерным или многомерным массивом, и т. п. Структура соответствует необходимому распределению данных в оперативной памяти узлов суперкомпьютера. При создании того или иного порта модуль определяет идентификаторы (имена) для всех входящих в него каналов, явно или неявно.

Связь между портами – это коммуникационный процесс, который обеспечивает передачу информации между двумя портами. Связываемые порты должны обладать одинаковой структурой и количеством каналов. Связь между портами реализуется как набор связей между соответствующими каналами этих портов, один к одному. Таким образом, когда пользователь устанавливает связь между портами, то на самом деле связываются каналы этих портов. Например, если порт X состоит из каналов $(x_1,x_2,..,x_N)$, а порт Y из каналов $(y_1,y_2,..,y_N)$, то связь между этими портами будет означать следующие связи каналов: $x_1 \rightarrow y_1$, $x_2 \rightarrow y_2$, ..., $x_N \rightarrow y_N$.

Работа модулей схемы подразумевается следующая. Модуль при запуске получает разные стартовые параметры. Среди параметров – список исполнителей. Модуль запускает свои вычислительные процессы на этих исполнителях. В контексте модуля определяется:

1. Какие и сколько вычислительных процессов следует запустить.
2. Какие параметры у этих процессов.
3. Глобальные идентификаторы для входных и выходных каналов процессов.

После запуска каждый вычислительный процесс организует чтение и запись своих каналов. Посредством связей каналы процесса передают данные другим процессам, и принимают от них данные. Отметим, что процесс может считывать сразу несколько каналов, и более того, выполнять реагирующие действие только когда во всех этих каналах подано очередное значение.

Алгоритм операции записи сообщения в канал.

1. При записи сообщения в канал пишущий процесс блокируется.
2. Проводится обход всех связей, в которых этот канал является источником. Для каждой связи вызывается операция записи сообщения в канал-приемник.
3. Проводится обход всех заявок от процессов на чтение этого канала.
4. Если читающий процесс работает на этом же исполнителе, то ему передается управление. Он выполнит свои реакции, и вернет управление, согласно структуре вычислительных процессов введенной ранее.
5. Если читающий процесс работает не на этом исполнителе, то сообщение пересылается ему методами межпроцессного или сетевого взаимодействия.
6. После обработки всех связей и заявок на чтение, управление возвращается процессу, который записал сообщение в канал.

Благодаря такому алгоритму, вычислительные процессы, находящиеся на одном исполнителе, имеют прямой и бесконфликтный доступ к памяти друг друга. Сообщение может содер-

жать указатели на блоки данных процесса. Поскольку пишущий процесс блокируется, то читающий процесс имеет полный доступ к его памяти, в том числе и на запись.

Шаг 4 алгоритма можно реализовать эффективно, сравнимо по затратам с вызовом функции. Таким образом передача сообщений между процессами на одном исполнителе, включая передачу больших блоков данных, будет производится быстро.

Описание метода схем завершено. Проиллюстрируем его применение на рис. 1–3.

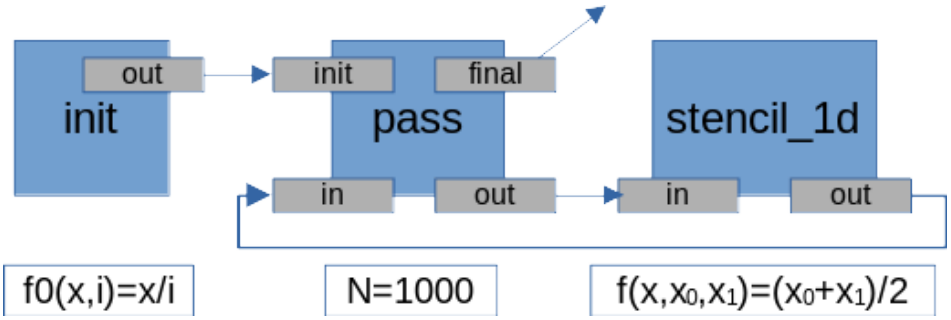

Рис. 1. Схема параллельной программы по вычислению функции одного переменного на регулярной сетке по явной схеме (см. раздел Испытание метода). Синие прямоугольники – модули, белые – параметры модулей, серые – порты, стрелки – связи портов. Модуль init генерирует начальные данные и записывает их в свой порт out. Модуль pass разово передает данные из порта init, а затем из порта in N раз – в порт out. Модуль stencil_1d реализует явную схему, вычисляя функцию f, заданную в параметре. Результат stencil_1d замкнут на модуль pass. В итоге работы схемы на порту final появляется результат вычисления N итераций

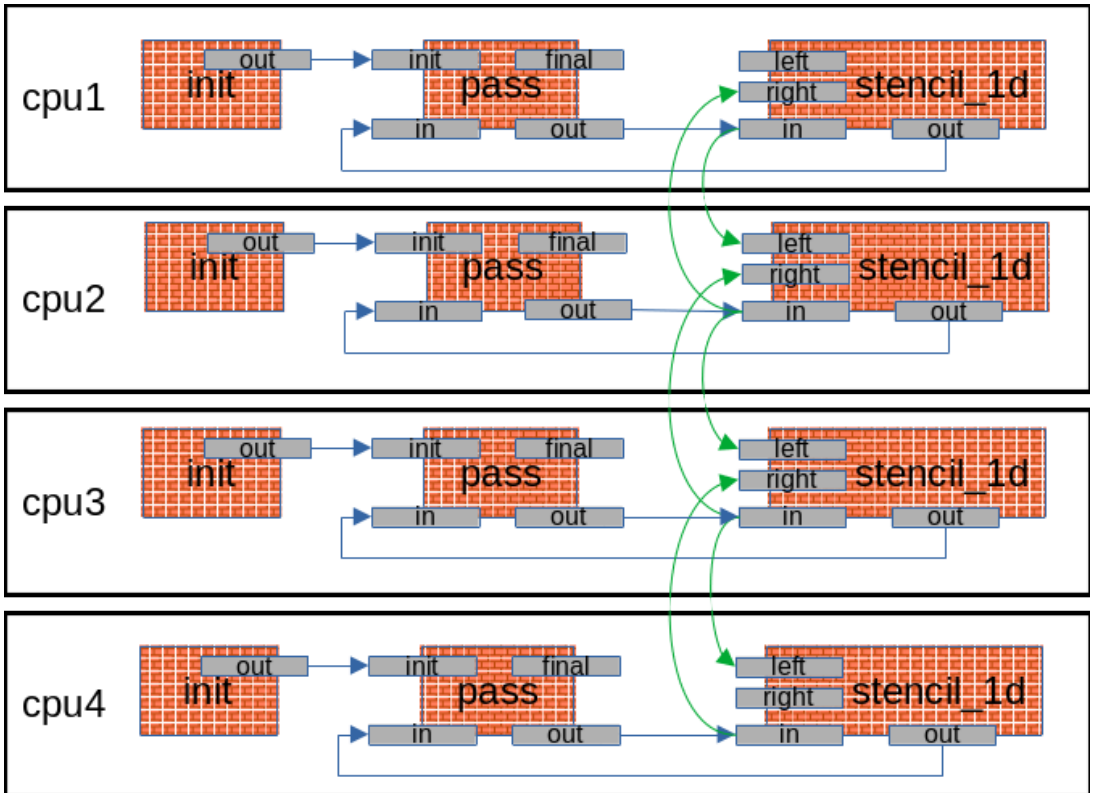

Рис. 2. Набор вычислительных процессов, порожденных схемой из рис. 1. Для иллюстрации показаны процессы для 4х исполнителей (cpu1..4). Кирпичные прямоугольники – вычислительные процессы, серые – каналы, стрелки – связи между каналами. Каждый исполнитель получил по процессу от каждого модуля. Исходя из связей портов образованы связи между каналами процессов. Также модуль stencil_1d наладил внутренние связи своих процессов для передачи граничных значений (зеленые стрелки)

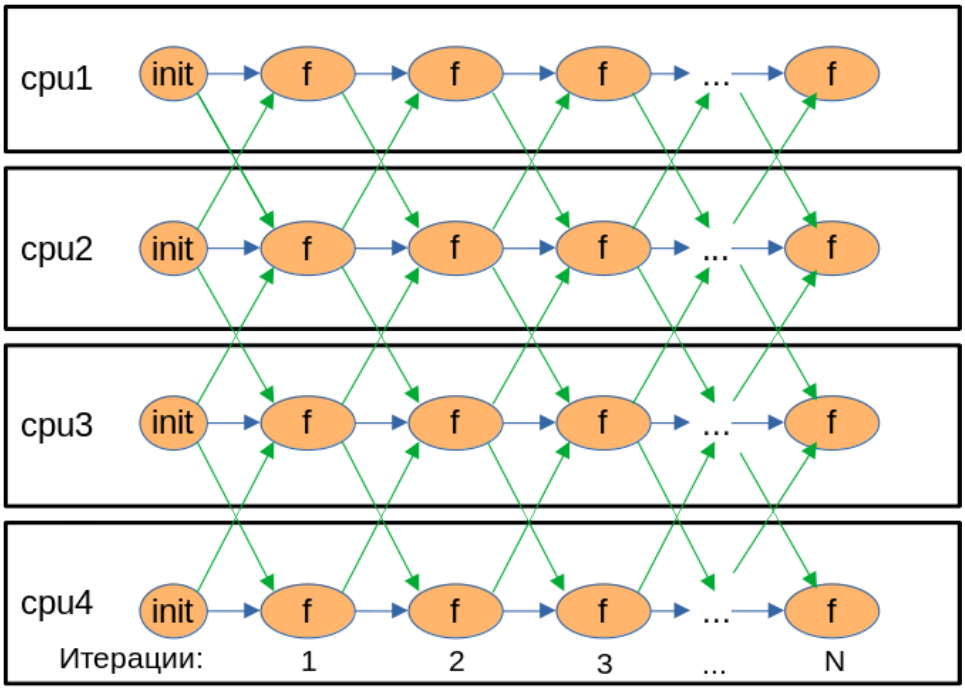

Рис. 3. Граф задач, порождаемый вычислительными процессами из рис. 2. Показано два типа задач: 1) init – задача формирует начальные данные, и 2) f – задача рассчитывает очередную итерацию на блоке данных. Задачи типа f «порождаются» и выполняются процессами stencil_1d, причем локально, на своих исполнителях

5.1 Связь схем и графов задач. Решение проблемы накладных расходов

Вычислительные процессы реагируют на поступление сообщений в каналах, выполняя те или иные алгоритмы. Эти алгоритмы (алгоритмы реакций на сообщения) по сути являются алгоритмами задач из модели графов задач. А процессы по сути являются генераторами задач.

Запуск процессов на исполнителях и связь их каналов определяет **распределенный** (в смысле без координирующего центра) процесс формирования графа задач. Сигналом к «добавлению» очередной задачи в граф служит сообщение в канале вычислительного процесса.

Добавленная таким образом задача тут же и «решается» – в алгоритме реакции на сообщение. Результаты ее работы записываются в исходящий канал, что приводит к срабатыванию реакций других вычислительных процессов – граф задач динамически развивается.

Таким образом, решается проблема накладных расходов **П1** на передачу задач. «Добавление» задач в граф задач в методе схем происходит неявно и распределенно, локально по отношению к исполнителям этих задач.

5.2 Прямой доступ к оперативной памяти процессов

Проблема **П2** – возможность прямого доступа к оперативной памяти с результатами выполнения задач – решена в представленном методе. Если два вычислительных процесса назначены на один исполнитель, то при передаче сообщения от первого процесса ко второму данные передаются по указателю, и могут быть свободно прочитаны или изменены. Благодаря зафиксированной структуре вычислительных процессов (см. выше) проблема одновременного доступа к памяти исключается.

5.3 Работа с блоками данных

В методе схем имеются следующие удобные условия:

1. Все блоки данных всех вычислительных процессов, размещенных в некотором исполнителе, находятся в оперативной памяти процесса ОС этого исполнителя.

2. Процессы в рамках одного исполнителя выполняются последовательно, обмениваясь данными через ссылки.

Исходя из этого, на текущий момент среди возможных вариантов просматривается следующее решение проблемы **П3**.

1. Вычислительный процесс может аллоцировать память у ОС для блока данных, который затем может отправить в исходящем сообщении.
2. После этой отправки начинается циркуляция блока данных между реакциями процессов текущего исполнителя. Она может быть продолжительной, например блок может передаваться в цикле между процессами (как в программе раздела 6). В конце концов такая циркуляция заканчивается, и блок становится «свободным». Процесс-аллокатор может использовать его повторно или вернуть ОС.
3. Допускается использовать входящий блок данных для записи результата вычисления реакции процесса. Это позволяет сократить объем используемой памяти. Для обеспечения этого требуются специальные меры, с приоритетом доступа для читающих процессов и одним пишущим процессом в конце.

5.4 Предварительная подготовка

Проблема **П4** по предварительной подготовке решается в представленном методе схем естественным образом. Вычислительный процесс может провести всю необходимую предварительную подготовку по загрузке библиотек, подготовке структур данных и т.п. И затем проводить неограниченное число вычислений на базе этих подготовленных сущностей. В этом смысле вычислительный процесс ведет себя также, как акторы из обзора литературы (см. выше).

6. Испытание метода

Для испытания метода проведен вычислительный эксперимент. Его материалы и коды программ опубликованы по адресу github.com/pavelvasev/ppk/tree/scdays2024/experiments/trial-v2. Впервые этот эксперимент был представлен в работе [23], в настоящей работе он дополнен.

Проводится итеративное вычисление:

- Вычисляется функция одного переменного на регулярной сетке по явной схеме.
- Используется формула: *p_next = (p_left + p_right)/2 + Math.random(1).*
- Это вычисление реализует упрощенную версию параллельного вычисления аппроксимации динамики нелокального уравнения неразрывности нелинейной марковской цепью [30]. Полная версия, выполненная на модели графов задач, описана в [15].

Сетка одномерная. Элементы сетки – числа с плавающей точкой типа float32.

Вычисление реализовано в последовательном варианте и в нескольких параллельных вариантах для разных моделей программирования:

- Последовательное вычисление (**seq**). Обычная последовательная программа.
- Граф задач (**task-graph**). Используется прямая реализация графа задач со статическим назначением задач на исполнители. Прямая реализация подразумевает, что пользователь запускает последовательную программу, которая связывается со средой вычисления и ставит задачи для графа задач посредством операций этой среды вычисления.
- Ручное распараллеливание (**manual**). Исполнители обмениваются данными посредством обмена сообщениями.
- Метод итераций для графа задач [23] (**iter-graph)**. Используется та же реализация графа задач, что и выше в методе графа задач (см. task-graph), для которой добавлены средства формирования подграфа итераций.
- Метод схем, представленный в настоящей работе (**schema).** Используется та же реализация графа задач, для которой добавлены средства формирования схем и их распространения на исполнители.

Параллельные программы используют один и тот же метод распараллеливания: сетка, как одномерный массив, разбивается на P одинаковых частей; на каждой итерации вычисление над

частями проводится параллельно; перед началом следующей итерации происходит обмен данными граничных значений этих частей.

Запуски всех программ выполнены как полная комбинация следующих параметров:

- Количество исполнителей – 2, 4, 8, 16. Исполнители работали на одной машине, каждому было выделено по одному ядру. Процессор Ryzen 1700x, 16 ядер.
- Размер вычислительной сетки: 100 тысяч, 1 миллион, и 10 миллионов. Применение небольшой сетки порождает «мелкозернистые» задачи, и зерна тем меньше, чем больше количество исполнителей.
- Количество итераций: 1 тысяча, 10 тысяч.

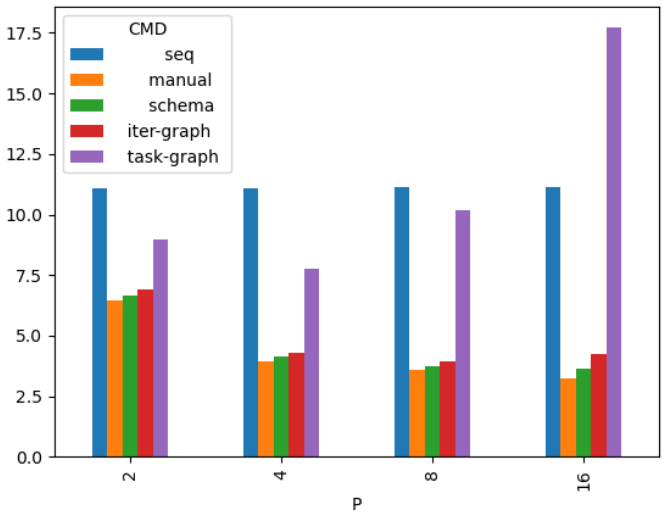

Рис. 4. Время работы программ на сетке 100 тысяч ячеек. По оси X – количество задействованных исполнителей, по оси Y – время в секундах. Цвет – модель программирования.

Сетка 100 тысяч ячеек дает наиболее мелкозернистые задачи, особенно с увеличением количества исполнителей. При 16 исполнителях зерно наиболее мелкое, и визуально заметна чувствительность моделей (а точнее их реализаций) к размеру зерна. Например, реализация графа задач показала производительность даже худшую, чем последовательное вычисление

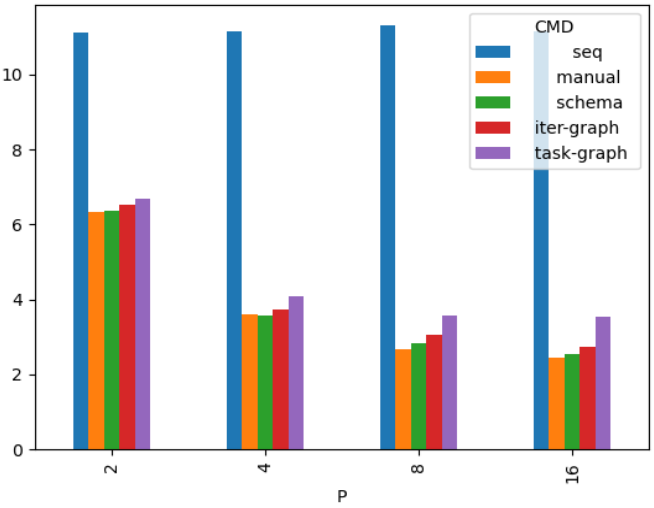

Рис. 5. Время работы программ на сетке 1 млн ячеек. По оси X – количество задействованных исполнителей, по Y – время в секундах. Цвет – модель программирования. Зерна задач крупнее, и время работы разных моделей вычислений стабилизировалось, по сравнению с зернами сетки 100 тысяч ячеек

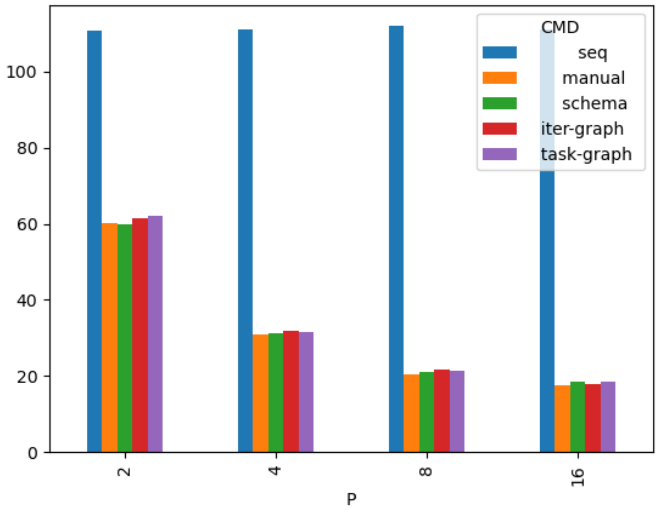

Рис. 6. Время работы программ на сетке 10 млн ячеек. По оси X – количество задействованных исполнителей, по Y – время в секундах. Цвет – модель программирования. Наблюдается схожесть работы методов во времени, зерна наиболее крупные и поэтому накладные расходы минимальны

Результаты испытания представлены на рис. 4–6. Выводы следующие:

1. Ручное распараллеливание показало себя наиболее эффективно. Это ожидаемо: программа была специально написана так, чтобы уменьшить накладные расходы.
2. Метод схем в среднем дал наиболее эффективное вычисление среди других испытанных методов автоматизации параллельного программирования, лучше остальных приближаясь по эффективности вычислений к ручному распараллеливанию.
3. Хорошо проявились накладные расходы при работе графа задач, что видно на рис. 4.
4. По видимому, есть диапазоны «зернистости» задач, где разные методы хороши или плохи. Другими словами, эффективность программ, сгенерированных методом, есть функция от размера (зерна) задачи.

В качестве перспектив дальнейших испытаний метода рассматриваются:

1. Более сложные в плане коммуникации задачи. Текущий тест относительно прост в плане коммуникации. Необходимы тесты на двумерных, трехмерных регулярных сетках; неструктурированных сетках; и на графах.
2. Существенно большее количество исполнителей, в том числе работающих одновременно на разных узлах суперкомпьютера.

7. Анализ метода и перспективы его развития

Принципиальные недостатки метода схем:

- **Статическое назначение на исполнители.** В методе подразумевается статическая балансировка нагрузки. Вместе с тем модель графов задач имеет возможность динамической балансировки, которая привносит преимущества, такие как распределение нагрузки исходя из текущей загруженности исполнителей, возможность гибко разделять суперкомпьютер между пользователями, и другие. Однако назначение задач, передача решений об этих назначениях по сети на исполнители, излишняя передача данных между задачами по сети вносят существенные задержки, которые и призван ликвидировать метод схем.
- **Неустойчивость к отказам узлов**. Для современных вычислений, а особенно это применимо к вычислениям экза-масштаба, уже получены оценки того, что вычислительные узлы будут выходить из строя часто, в том числе во время вычислений [25]. Метод схем в своей постановке не учитывает этой проблемы. И пока нет понимания, возможна ли

разработка эффективного дополнения, которое бы обеспечило отказоустойчивость в этой ситуации.

Устранимые недостатки метода схем:

- **Сложность асинхронного взаимодействия схем.** Метод схем удобен для вычислений, когда данные между процессами передаются на регулярной основе. Однако существуют ситуации, когда передача данных требуется лишь иногда. Например, это обычная ситуация в сценарии онлайн-визуализации [26]. При такой визуализации пользователь может запросить данные в произвольный момент времени. Технически это означает, что данные из некоторых модулей основной вычислительной схемы по внешней команде необходимо передать на входные порты некоторой схемы визуализации. При этом необходимо обеспечить, чтобы данные были переданные согласованно, из соответствующих вычислительных итераций. Это удается реализовать, но требует специальных мер, основанных на подсчете сообщений в заданных каналах, т. е. номеров итераций.

Среди преимуществ метода схем выделяются следующие:

- **Высокая производительность вычислений.** По сути, метод обеспечивает наполнение вычислительных процессов уровня ОС (процессы исполнителей) путем стыковки алгоритмических слоев, определяемых вычислительными процессами модулей. Эта стыковка обеспечивает передачу данных между этими слоями путем передачи управления в рамках одного процесса ОС, что эквивалентно вызову функций с передачей данных по указателям. Что по эффективности эквивалентно тому, как если бы эти процессы программировались вручную.
- **Динамическое управление содержанием вычисления.** Схему можно менять во время выполнения, и это влечет изменение состава выполняющихся вычислительных процессов в процессах ОС исполнителей, и меняет их взаимодействие между собой.
- **Возможность абстрагирования схем.** Написанную схему можно зафиксировать в качестве нового модуля. Таким образом, метод схем отвечает всем критериям выразительности языков программирования [27]. В нем есть базовые элементы (модули), средства их комбинирования (связи между портами модулей), и операция абстрагирования.
- **Удобное мета-программирование.** Программа, формирующая схему, может быть последовательной, что удобно для мышления. Она может быть написана на удобном для работы языке, например на Python или даже в форме разметки типа YAML. При этом вычислительные процессы, которыми схема управляет, могут быть оптимизированные, например написанные на Си или для GPU.

Среди перспектив развития метода, помимо устранения упомянутых выше недостатков, представляются полезными следующие:

1. Более строгая модель работы с памятью блоков данных (раздел 5.3).
2. Ввод модели сериализации объектов. Если объект передается вычислительному процессу в том же процессе ОС, то его можно передать по указателю. Если же объект передается по сети, то его необходимо сериализовать (в исходящий поток согласно используемой технологии передачи данных) и затем десериализовать при получении. Эти аспекты должны быть проработаны. Сейчас это решается таким образом, что передаются объекты, заранее согласованные со средой выполнения, такие как массивы чисел.
3. Автоматизация назначения процессов модулей на исполнители: сейчас это делается «вручную», путем указания исполнителей модулям. Здесь видится большая перспектива, в том числе на основе сбора статистики, искусственного интеллекта и т.д.
4. Возможность подключения программ, заданных методом схем, к параллельным программам, написанным в других моделях вычислений. Для этого достаточно вынести абстракцию «порта» на внешний уровень, как это сделано в ADIOS2 [29].
5. Повышение эффективности реализации, в частности использование суперкомпьютерных технологий передачи данных (InfiniBand, разные RDMA), например посредством библиотеки UCX.

Вероятно не все эти возможности стоит встраивать в сам метод, но стоит предложить такой программный интерфейс, который позволял бы добавлять их внешним образом.

8. Заключение

Модель графов задач автоматизирует программирование процессов, составляющих параллельные программы. Например, в технологии MPI эти процессы программируются вручную, и программист для каждого процесса описывает, что и когда он делает (когда считает, когда посылает сообщения, что делает при получении сообщений и т. п.). В модели графов задач эти же самые процессы формируются автоматически, исходя из тех задач что ставятся управляющим алгоритмом. При этом управляющий алгоритм может быть последовательным, что существенно упрощает программирование параллельной программы.

Но оказывается, что при подобной автоматизации возникают существенные накладные расходы. Эти расходы при определенных ситуациях оказываются таковы, что параллельная программа может работать даже медленнее, чем последовательная (см. пример на рис. 4).

В настоящей работе предложено возможное решение, как избежать этих накладных расходов. Предлагается использовать «метод схем». Его суть заключается в том, что пользователь описывает параллельную программу как набор взаимодействующих модулей. Каждый такой модуль определяет вычислительные процессы, которые запускаются в процессах уровня ОС на узлах суперкомпьютера. Взаимодействие модулей определяется связями между их портами. Эти связи в свою очередь формируют связи между вычислительными процессами, а точнее между их каналами коммуникации. Таким образом описание схемы, составленное из модулей и связей между ними, определяет поведение комплекса процессов ОС на суперкомпьютере. При этом внутри этих процессов, на теоретическом уровне, работает модель графов задач.

Метод схем является абстракцией более высокого уровня, чем графы задач. По сути он является другой моделью вычислений, со своими сущностями и правилами их взаимодействия. Полагается, что эта модель сможет эффективно порождать динамический граф задач, избегая при этом обременительных накладных расходов.

Отметим связь метода схем с моделью акторов, применяемую например в системе HPX, Charm++ и других. Актор соответствует вычислительному процессу. Модуль схемы определяет свои вычислительные процессы, то есть как бы задает целую группу акторов. Эта группа взаимодействует с другими группами акторов, заданных другими модулями схемы. Взаимодействие определяется связями портов, и как следствие связями каналов «акторов»[9].

Получается, что метод схем есть способ управления «ансамблем из групп акторов».

Отметим идейную связь предложенного метода с промышленными системами. В проекте Логос-Платформа [28] предлагается модель организации междисциплинарных вычислений на основе связи вычислительных программ посредством внедрения адаптеров (пользовательских функций) программ друг в друга. При этом программы могут быть написаны в произвольных моделях программирования. Метод схем организует подобную связь на уровне модулей. Однако у модулей в настоящей работе нет возможности реагировать на подключение их портов к другим модулям, что необходимо для рассылки адаптеров, и вероятно эту возможность следует предусмотреть.

В проекте ADIOS2 [29] моделей вычислений не предлагается, но предлагается способ связи параллельных программ. Этот способ соответствует предложенному понятию связи между портами модулей. В ADIOS2 к связям добавлена модель времени для итеративных вычислений (beginStep/endStep), в предложенном методе этого пока нет. Связи программ в ADIOS2 многофункциональны, например могут поддерживать перераспределение данных (MxN data transfer). В методе схем это достигается явно, посредством ввода перераспределяющего модуля.

Авторы благодарят коллег, в общении с которыми и сформировалась настоящая работа. Особую благодарность авторы выражают за постановку задачи [30], которая сыграла роль мотива представленного исследования.

[9]Взаимодействие посредством связей каналов обеспечивает лучшую гибкость, то есть настраиваемость системы, чем адресная посылка сообщений и удаленный вызов процедур, которые часто применяются в программировании.

Литература

1. Котов В.Е., Нариньяни А.С. Асинхронные вычислительные процессы над памятью // Кибернетика. 1966. № 3. С. 64–71.
2. Котов В.Е. Проблемы развития параллельного программирования // Труды Всесоюзного симпозиума «Перспективы системного и теоретического программирования». Новосибирск, 1979. С. 58–72.
3. Karp R.M., Miller R.E. Parallel program schemata // J. Comp. Syst. Sci. 1969. Vol. 3, no. 2. P. 147–195. DOI: 10.1016/S0022-0000(69)80011-5.
4. Mikloško J., Kotov V.E. Algorithms, Software and Hardware of Parallel Computers. Springer, Berlin, Heidelberg, 1984. 395 p. DOI: 10.1007/978-3-662-11106-2.
5. Einar S. Non-Sequential Processes and Concurrency Theory // Carl Adam Petri: Life and Science. 2015. P. 77–87. DOI: 10.1007/978-3-662-48093-9.
6. Kahn G. The semantics of a simple language for parallel programming // Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974. P. 471–475.
7. Ершов А.П. Смешанные вычисления // В мире науки. 1984. № 6.9.
8. Ахмед-Заки Д.Ж., Лебедев Д.В., Малышкин В.Э., Перепёлкин В.А. Автоматизация конструирования распределенных программ численного моделирования в системе LuNA на примере модельной задачи // Проблемы информатики. 2019. № 4 (45). С. 53–64. DOI: 10.24411/2073-0667-2019-00017.
9. Blass A., Gurevich Y. Abstract state machines capture parallel algorithms: Correction and extension // ACM Trans. Comput. Log. 2008. Vol. 4, no. 4. P. 578–651. DOI: 10.1145/937555.937561.
10. Куприянов Б.В. Оценка и оптимизация производительности рекурсивного конвейера // Автоматика и телемеханика. 2020. Т. 5. С. 6–25.
11. Воеводин В.В., Воеводин Вл.В. Параллельные вычисления. СПб.: БХВ-Петербург, 2002. 608 с.
12. Гергель В.П., Стронгин Р.Г. Основы параллельных вычислений для многопроцессорных вычислительных систем: Учебное пособие. Нижегородский гос. ун-т им. Н. И. Лобачевского. Нижний Новгород: Издательство Нижегородского госуниверситета, 2001. 122 с.
13. Прихожий А.А. Распределенная и параллельная обработка данных // Белорусский национальный технический университет, кафедра «Программное обеспечение вычислительной техники и автоматизированных систем». Минск: БНТУ, 2016. 91 с.
14. Stevenson S., Jones S., Oh F. Enabling Dynamic Control Flow in CUDA Graphs with Device Graph Launch. 2022. Nvidia developer blog. URL: https://developer.nvidia.com/blog/enabling-dynamic-control-flow-in-cuda-graphs-with-device-graph-launch/.
15. Васёв П.А. Визуализация работы алгоритма планирования параллельных задач // Труды 33-ей Международной конференция по компьютерной графике и машинному зрению ГрафиКон 2023, 19-21 сентября 2023 г., Москва. С. 341–353. DOI: 10.20948/graphicon-2023-341-353.
16. Васёв П.А. Гибридное планирование графов задач // Сборник тезисов докладов 12-го Национального суперкомпьютерного форума НСКФ-2023, Россия, Переславль-Залесский, ИПС имени А.К. Айламазяна РАН, 28 ноября – 1 декабря 2023 года.
17. Vasev P. A Computational Model for Interactive Visualization of High-Performance Computations // Supercomputing. RuSCDays 2023. Vol. 14389 / eds. by V. Voevodin, S. Sobolev, M. Yakobovskiy, R. Shagaliev. Springer, Cham, 2023. Lecture Notes in Computer Science. DOI: 10.1007/978-3-031-49435-2_9.
18. Castes C., Agullo E., Aumage O., Saillard E. Decentralized in-order execution of a sequential task-based code for shared-memory architectures. Research Report RR-9450, Inria Bordeaux - Sud Ouest. 2022. 30 p.

19. Hoque R., Herault T., Bosilca G., Dongarra J. Dynamic task discovery in PaRSEC: a data-flow task-based runtime // Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA '17). Association for Computing Machinery, New York, NY, USA, 2017. Article 6. P. 1–8. DOI: 10.1145/3148226.3148233.
20. Арыков С.Б., Малышкин В.Э. Алгоритмы конструирования асинхронных программ заданной степени непроцедурности методом группировки // Вестник Новосибирского государственного университета. Серия: Информационные технологии. 2009. Т. 7, № 1. С. 3–15.
21. Арыков С.Б. Асинхронная модель вычислений над общей памятью // Вестник Уфимского государственного авиационного технического университета. 2016. Т. 20, № 3 (73). С. 114–121.
22. Cambier L., Qian Y., Darve E. TaskTorrent: a Lightweight Distributed Task-Based Runtime System in C++ // 2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM), Atlanta, GA, USA, 2020. P. 16–26. DOI: 10.1109/PAWATM51920.2020.00007.
23. Васёв П.А. Метод итераций для графов задач // Параллельные вычислительные технологии – XVIII всероссийская конференция с международным участием, ПаВТ'2024, г. Челябинск, 2-4 апреля 2024 г. Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2024. С. 182. DOI: 10.14529/pct2024.
24. Lohstroh M., Menard C., Bateni S., Lee E.A. Toward a Lingua Franca for Deterministic Concurrent Systems // ACM Trans. Embed. Comput. Syst. 2021. Vol. 20, no. 4. Article 36. DOI: 10.1145/3448128.
25. Четверушкин Б.Н., Якобовский М.В. Вычислительные алгоритмы и архитектура систем высокой производительности // Препринты ИПМ им. М.В.Келдыша. 2018. № 52. 12 с. DOI: 10.20948/prepr-2018-52.
26. Vasev P. Analyzing an Ideas Used in Modern HPC Computation Steering // 2020 Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT), Yekaterinburg, Russia, 2020. P. 1–4. DOI: 10.1109/usbereit48449.2020.9117685.
27. Abelson H., Sussman G.J., Sussman J. The Elements of Programming // Structure and Interpretation of Computer Programs. MIT Press/McGraw-Hill, Cambridge, 1996.
28. Надуев А.Г., Черевань А.Д., Лебедева А.С. Архитектура программного модуля «ЛОГОС ПЛАТФОРМА» // Вопросы атомной науки и техники. Серия: Математическое моделирование физических процессов. 2022. № 4. С. 55–63. DOI: 10.53403/24140171_2022_4_55.
29. Klasky S. et al. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management // SoftwareX. 2020. Vol. 12. DOI: 10.1016/j.softx.2020.100561.
30. Averboukh Y. Lattice approximations of the first-order mean field type differential games // Nonlinear Differ. Equ. Appl. 2021. Vol. 28, no. 65. DOI: 10.1007/s00030-021-00727-2.
31. Киреев С.Е., Литвинов В.С. Анализ исполнения фрагментированных программ на основе факторов SLOW // Параллельные вычислительные технологии – XVIII всероссийская конференция с международным участием, ПаВТ'2024, г. Челябинск, 2-4 апреля 2024 г. Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2024. С. 94–107. DOI: 10.14529/pct2024.
32. Durner D., Leis V., Neumann T. On the Impact of Memory Allocation on High-Performance Query Processing // Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN'19). ACM, 2019. Article 21. P. 1–3. DOI: 10.1145/3329785.3329918.
33. Васёв П.А. Среда параллельного программирования Parallel Programming Kit // Тезисы докладов Национального суперкомпьютерного форума (НСКФ-2024), 26-29 ноября 2024, Институт программных систем им. А.К. Айламазяна РАН, Переславль-Залесский.
34. Marquer Y., Gava F. Algorithmic Completeness of BSP Languages. Technical Report Laboratoire d'Algorithmique, Complexité et Logique, Université Paris-Est Créteil. 2018.