ISSN 1992-6502 (Print) Вестник УГАТУ ISSN 2225-2789 (Online)
2016. Т. 20, № 1 (71). С. 175–186 http://journal.ugatu.ac.ru

УДК 004.65

РАСПРЕДЕЛЕННАЯ ОБРАБОТКА ДАННЫХ В ЗАДАЧАХ МОДЕЛИРОВАНИЯ ТЕРМОДИНАМИЧЕСКОГО РАВНОВЕСИЯ В МИКРОСИСТЕМАХ ГАЗ-МЕТАЛЛ МЕТОДАМИ МОЛЕКУЛЯРНОЙ ДИНАМИКИ

Д.В. Пузырьков[1], В.О. Подрыга[2], С.В. Поляков[3,4]

[1] dpuzyrkov@gmail.com, [2] pvictoria@list.ru, [3] polyakov@imamod.ru

[1,2,3] Федеральное государственное учреждение «Федеральный исследовательский центр Институт прикладной математики им. М. В. Келдыша Российской академии наук» (ИПМ им. М. В. Келдыша РАН)
[4] Национальный исследовательский ядерный университет МИФИ (НИЯУ МИФИ)

Поступила в редакцию 01.03.2016

Аннотация. В данной работе представлен программный комплекс для распределенной обработки и визуализации результатов моделирования задач молекулярной динамики. В процессе создания комплекса были изучены вопросы производительности интерпретируемого языка Python и средства, позволяющие ее увеличить. Также были исследованы проблемы получения и обработки данных, находящихся на множестве вычислительных узлов. В качестве средства визуализации был выбран открытый пакет для научной визуализации "Mayavi2". В результате применения комплекса в исследовании взаимодействия газа с металлической пластиной удалось в деталях наблюдать эффект адсорбции, который важен для многих практических приложений.

Ключевые слова: распределенная обработка данных; визуализация; молекулярная динамика; облачные вычисления; параллельные алгоритмы; Python; Mayavi2; Numba.

ВВЕДЕНИЕ

Развитие компьютерных технологий и быстрый рост вычислительных мощностей в настоящее время существенно расширили возможности вычислительного эксперимента (ВЭ). В частности, на современном этапе уже удается изучать свойства и процессы в сложных системах на молекулярном и даже атомарном уровнях, например, методами молекулярной динамики (МД). Математические модели такого типа могут оперировать огромным количеством частиц: от миллионов до миллиардов и более. При этом каждая частица может описываться десятками параметров. Объемы выходных данных в подобных ВЭ могут исчисляться терабайтами информации. В настоящее время наиболее используемым подходом для ускорения масштабных вычислений является их распараллеливание, когда множество вычислительных узлов обрабатывает большой массив данных по частям. В результате распараллеливания каждый узел получает лишь небольшую локально расположенную часть данных, которую легко обработать. Данная методика существенно уменьшает время, необходимое для полной обработки данных, но приводит к нескольким проблемам, связанным с их хранением. Чаще всего после выполнения расчетов вычислительные узлы обмениваются результирующими данными, и мастер-процесс сохраняет их либо в собственной оперативной памяти, либо на диске в виде одного большого массива или файла. Однако в больших задачах размер массива (файла) может существенно превысить ресурсы мастер-узла. В таком случае каждый вычислительный узел хранит полученные результаты обособленно. Этот способ хранения дает несколько преимуществ. Первое из них – отсутствие необходимости последовательного чтения всех результатов для последующей обработки (например, в целях визуализации), поскольку каждый вычислительный узел считывает только свои данные. Второе – каждый отдельный файл данных

Работа выполнена при поддержке РФФИ (гранты № 15-07-06082-а, № 15-29-07090-офи_м). Статья рекомендована к публикации программным комитетом международной научной конференции «Параллельные вычислительные технологии 2016»

обычно не очень велик (по сравнению с полным массивом), и, следовательно, его обработка будет занимать меньше времени. Расчетные программы такого типа и рассматриваются в данной статье, а именно, алгоритм и его реализация, частично обсуждавшиеся в работе [1].

Одним из способов представления данных ВЭ является двух- и трехмерная визуализация. Для того чтобы построить общую картину результатов моделирования, требуется прочитать и обработать данные с каждого вычислительного узла, что само по себе является ресурсоемкой задачей.

В большинстве случаев форматы расчетных данных и способ их хранения сильно отличаются в зависимости от расчетной программы. Поэтому такие программы либо имеют свой визуализатор и рассчитывают все нужные для визуализации характеристики в процессе работы, собирая их потом на мастер-узле и отрисовывая своими средствами (LAMMPS и другие), либо сохраняют данные в общеизвестные стандартизированные контейнеры (HDF5, VTK и другие), которые поддерживаются большинством программных продуктов для научной визуализации. Проблемы таких способов хранения и отрисовки проявляются в ограниченности возможностей собственного программного обеспечения расчетного комплекса в отношении визуализации и постобработки, а в случае общеизвестных стандартов хранения данных это проблема большого размера таких файлов.

В данной работе представлена попытка создать программный комплекс, позволяющий в интерактивном режиме импортировать, обрабатывать и визуализировать данные из разных источников, будь то данные известных форматов или распределенные по вычислительным узлам результаты расчетов в пользовательском формате.

В качестве тестового примера были рассмотрены результаты, полученные с помощью вычислительной программы, описанной в работе [1]. Ввиду параллельности алгоритма и особенностей способа хранения эти данные могут представлять собой как один большой файл, описывающий общее состояние симулируемой системы, так и распределенные по вычислительным узлам его части, над которыми работал каждый вычислительный процесс по отдельности. Данные, полученные в результате моделирования, представляют собой информацию о взаимодействиях молекул газа с атомами металла вблизи его поверхности. Подобный процесс характерен для многих технических микросистем, использующихся в нанотехнологии.

ПОСТАНОВКА ЗАДАЧИ

Задача сбора и обработки распределенных данных, полученных в результате выполнения некоторой расчетной программы, имеет несколько ключевых особенностей. Во-первых, это конкретика предметной области задачи. В результате поисков среди различных пакетов моделирования не было найдено подходящих средств для параллельной загрузки распределенных данных, относящихся к расчетной задаче. Это и послужило поводом для данной разработки. Во-вторых, это размер входных данных. Они могут быть как небольшим одномерным массивом, так и большим количеством файлов, распределенных по различным вычислительным узлам и файловым системам. Такие задачи обычно решаются либо средствами программного комплекса, породившего эти данные, либо разработкой специализированного обработчика, который "понимает" форматы ввода-вывода используемой расчетной программы. В-третьих, это необходимость обрабатывать такие результаты для удобного представления на графиках или в 3D-визуализации.

В связи с этими особенностями в данной работе была предпринята попытка создать каркас для системы, обладающей следующими возможностями:

- параллельное чтение данных из разных источников;
- определение пользовательских форматов данных;
- определение пользовательских фильтров и обработчиков;
- предоставление средства для визуализации данных.

Важно подчеркнуть, что в случае разработки такой библиотеки большую роль играет ее расширяемость, позволяющая относительно просто использовать эту систему для обработки данных, хранящихся в любом формате.

В качестве начального этапа разработки была выбрана задача постобработки и визуализации данных, полученных в работе [1].

Эта задача подразумевает рассмотрение всех перечисленных особенностей выбранного приложения, так как результаты его вычислений распределены по различным вычислительным системам с удаленным доступом к ним по SSH.

СРЕДСТВА РАЗРАБОТКИ

Ввиду необходимости расширения в будущем создаваемой программной системы было решено использовать для ее реализации интер-

претируемый язык программирования. В качестве такого языка был выбран Python.

Python [2] – интерпретируемый язык программирования, широко используемый в научном сообществе. Его дизайн настраивает программиста на написание хорошо читаемого кода, а его синтаксис позволяет описывать алгоритмы за меньшее число строк, чем это возможно на C++ или Java. Python также является кроссплатформенным, что позволяет использовать его (с некоторыми ограничениями) как под ОС MS Windows, так и под ОС Linux. Синтаксис ядра Python очень прост и краток. В то же время его стандартная библиотека предоставляет огромный объем полезных инструментов, а также удобные структуры данных. Python поддерживает несколько парадигм программирования, в том числе объектно-ориентированную, императивную и функциональную. Также этот язык имеет динамическую "утиную" типизацию и автоматическое управление памятью. Хотя Python уже имеет версию 3, в данной работе использовалась версия Python 2.7, ввиду того, что некоторые используемые библиотеки (например, Mayavi2) были написаны на Python 2.7, а Python 3 и Python 2.7 в некоторых случаях обратно не совместимы.

IPython [3] – интерактивная оболочка для языка Python, добавляющая расширенную интроспекцию, дополнительный командный синтаксис, подсветку кода и автоматическое дополнение. Основная особенность этого проекта в том, что он предоставляет ядро для web-приложения Jupyter, позволяющего писать скрипты на языках Python, R и BASH прямо в браузере, а также взаимодействовать с объектами визуализации. В данной работе приложение IPython notebook было выбрано в качестве системы web-управления.

УСКОРИТЕЛИ ВЫЧИСЛЕНИЙ

Несмотря на все достоинства, у основной реализации интерпретатора, CPython, имеется достаточно большой минус, связанный со скоростью выполнения и многопоточностью. Последнее обусловлено использованием механизма GIL (Global Interpreter Lock), представляющего собой mutex (простейший двоичный семафор), не позволяющий разным потокам выполнять один и тот же байткод одновременно. Эта блокировка, к сожалению, является необходимой, так как система управления памятью в CPython не является потокобезопасной. Для обхода этого ограничения были рассмотрены следующие способы.

Numpy [4] – библиотека с открытым исходным кодом, реализующая параллельную (векторизованную) поддержку многомерных массивов и методов работы с ними. Так как Numpy написана на языке C, то исполняемый код библиотечных алгоритмов является скомпилированным в машинный код, и не возникает необходимости в его повторной интерпретации, что ускоряет исполнение конкретных функций. Также потоки, запускаемые внутри Numpy, не зависят от GIL, присутствующего в CPython, а следовательно, их использование ускоряет исполнение алгоритмов за счет распараллеливания вычислений. К тому же Numpy имеет подробную документацию, что облегчает процесс разработки и поддержки ПО. Все эти особенности делают Numpy обоснованным выбором для вычислений на языке Python.

Numba [5] представляет собой оптимизирующий Just-In-Time (JIT) компилятор, позволяющий ускорять критические ко времени выполнения участки программы, компилируя их в машинный код. В отличие от Cython, Numba не требует явной аннотации типов (хотя поддерживает такую возможность) и не транслирует код в язык C, что упрощает использование этой технологии. Для того, чтобы определить, какие методы программы требуется оптимизировать, пользователь должен использовать простейшее средство языка Python, называемое декоратором. Помеченные соответствующим декоратором методы Numba оптимизирует и компилирует в машинные коды, используя инфраструктуру LLVM (Low Level Virtual Machine). Благодаря возможностям отключения GIL, а также компиляции в машинный код, не использующий Python C API (для методов, оперирующих простейшими типами), компилятор Numba может генерировать более эффективный и оптимизированный байткод. Также Numba старается автоматически векторизовать все, что может, используя возможности многопроцессорных систем по максимуму.

В табл. 1 приведено сравнение скорости выполнения одного и того же Python кода (перемножение массивов с умножением и делением на константу, рис. 1), в одном случае без использования Numba, в другом используя эту технологию. Тестирование проводилось на системе с ЦПУ Intel Core I7.

Стоит заметить, что описание алгоритма, приведенного на рис. 1, не является параллельным, то есть Numba позволяет ускорить вычислительную часть кода почти в два раза за счет JIT компиляции, не прибегая ни к каким особым

оптимизациям, как, скорее всего, пришлось бы сделать при использовании любого другого средства.

Таблица 1

Сравнение производительности Numpy и Numba

N	Numpy	Numba + Numpy	Отн. прирост производительности
10^6	0.19 мс	0.07 мс	2.77
10^7	1.62 мс	0.74 мс	2.19
10^8	16.06 мс	7.4 мс	2.17

СРЕДСТВА ПАРАЛЛЕЛИЗАЦИИ

Ввиду наличия у CPython такого механизма, как GIL, использование стандартных потоков Python не является эффективным решением для параллельной обработки данных. GIL не позволяет выполняться одновременно нескольким потокам (в рамках одного процесса интерпретатора) даже на многопроцессорной системе. Однако запуск нескольких процессов-интерпретаторов, обменивающихся данными, вполне решает эту проблему. Единственная особенность в данном случае состоит в том, что запуск процесса – намного более долгая операция, чем запуск потока, и использовать многопроцессное приложение на небольших данных нерационально.

Существует несколько инструментов для удобного управления такими задачами.

Multiprocessing [2] **–** модуль стандартной библиотеки Python, предоставляющий интерфейс для создания и управления множеством процессов интерпретаторов. Его API похож на модуль threading из стандартной библиотеки. Кроме этого, он добавляет некоторый новый функционал, например класс Pool, представляющий собой абстракцию и управляющий механизм для набора параллельных процессов интерпретатора. Multiprocessing также реализует межпроцессные примитивы, такие как очередь и mutex. Также стоит отметить, что каждый процесс интерпретатора работает в раздельной оперативной памяти, следовательно, не требуется беспокоиться о состояниях гонки при записи или чтении переменных, если только они не объявлены как объект в разделяемой памяти.

Обмен данными между процессами интерпретатора внутри данной библиотеки происходит через канал межпроцессного взаимодействия, основанного на неименованных каналах (pipes), с использованием модуля pickle, позволяющего "сериализовывать" и "десериализовывать" Python-объекты (сериализация – процесс перевода какой-либо структуры данных в последовательность битов; десериализация – восстановление начального состояния структуры данных из битовой последовательности). Все задачи по синхронизации и пересылке объектов берет на себя модуль Multiprocessing. Поэтому пользователю не нужно решать задачу подтверждения того, что все используемые вычислителем данные были обновлены.

ParallelPython [6] **–** библиотека для решения проблемы кластеризации приложения. Ее реализация имеет клиент-серверную структуру и требует установки серверной части на вычислительные узлы. Из-за простоты своего интерфейса ParallelPython позволяет практически в одну строку запустить выполнение некоторой определенной задачи на множестве узлов, объединенных сетью. Данная библиотека имеет свой балансировщик загрузки, и также сама отслеживает состояние узлов и перераспределяет задачи в случае недоступности одного из них. Вместе с модулем Multiprocessing ParallelPython позволяет достаточно просто и удобно использовать все возможности вычислительных кластеров.

На рис. 2 представлен пример суммирования множества массивов в параллельном режиме с использованием ParallelPython и Multiprocessing. На каждом вычислительном узле запускается два процесса с помощью ParallelPython, и каждый из них запускает еще два процесса средствами Multiprocessing. Стоит заметить, что данная библиотека, так же как и Multiprocessing, использует модуль pickle для сериализации данных и tcp/ip сеть для обмена сообщениями.

```
from numba import jit
@jit(nogil=True, nopython=True)
def numpy_numba_func(vx, vy, vz, multiplier=100, divider=3.0):
       return multiplier*((vx*vx) + (vy*vy) + (vz*vz)) / divider

def numpy_func(vx, vy, vz, multiplier=100, divider=3.0):
    return multiplier*((vx*vx) + (vy*vy) + (vz*vz)) / divider
```

Рис. 1. Сценарий для сравнения производительности перемножения массивов средствами Numpy и Numba

СРЕДСТВА ВИЗУАЛИЗАЦИИ

Существует множество сторонних средств для визуализации данных. Для примера можно рассмотреть Paraview, VMD, Aviso, Tecplot и др. Каждый из перечисленных программных комплексов имеет собственный формат хранения данных, а также умеет читать стандартизованные форматы. Представленную в данной работе библиотеку можно использовать как средство для подготовки данных для последующей визуализации в перечисленных пакетах, однако также было решено добавить в нее свои средства визуализации. В результате исследования оказалось, что перечисленные ниже библиотеки почти не уступают в возможностях большим пакетам для визуализации. В качестве таких библиотек для визуализации общего назначения были выбраны следующие.

Mayavi2 [7] **–** библиотека для языка Python, позволяющая строить визуализации общего назначения. Она предоставляет пользователю возможности для загрузки и отрисовки данных в отдельном графическом приложении, а также удобный Python API для программной генерации изображений. Данная библиотека построена поверх известной в научных кругах библиотеки VTK.

Mayavi2 предоставляет широкие возможности для визуализации данных, начиная от гидродинамических расчетов и заканчивая атомистическими данными. В случае интерактивного использования также доступны инструменты для изменения параметров визуализации на лету, таких как размеры объектов, цветовые схемы, параметры фильтров, добавление на сцену текста и заголовков. Mayavi2 имеет также возможность offscreen-отрисовки (в оперативной памяти), что чрезвычайно важно для серверной, распределенной и пакетной обработки большого количества данных.

На рис. 3 представлен пример расчета плотности распределения точек и ее трехмерная визуализация с использованием Mayavi2 и библиотеки для научных вычислений SciPy.

Matplotlib [8] **–** Python-библиотека для построения качественных двухмерных графиков. Она широко используется в научном сообществе. Использование Matplotlib очень похоже на использование методов построения графиков в MATLAB, однако это независимые проекты.

Особенно удобно, что графики, отрисовываемые с помощью этой библиотеки, можно легко встраивать в приложения, написанные с использованием различных библиотек для построения графического интерфейса. Matplotlib может встраиваться в приложения, написанные с использованием библиотек wxPython, pyQT и pyGTK.

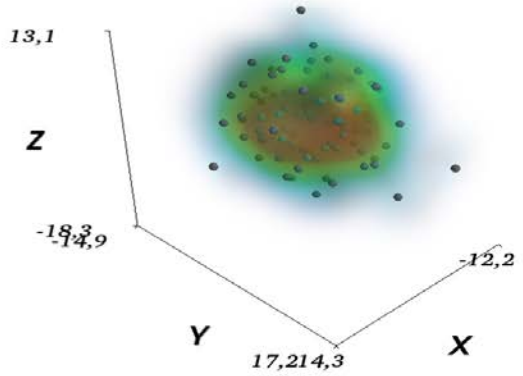
Модуль Matplotlib не входит в стандартную библиотеку, но является стандартом де-факто при визуализации числовой информации.

ДОСТУП К РАСПРЕДЕЛЕННЫМ ДАННЫМ

Данные, полученные в работе [1], имеют распределенную структуру и хранятся на вычислительных узлах, использовавшихся для симуляции. На рис. 4 показана примерная схема расположения таких данных. Совокупность всех файлов представляет собой полное состояние молекулярно-динамической системы в конкретный момент времени. Бывает, что вычислительные узлы используют общее дисковое пространство, например, используя NFS. Однако

```
import pp
import numpy as np
ppservers = ("10.0.0.1","10.0.0.2","10.0.0.3","10.0.0.4")
serv = pp.Server(ncpus = 2, ppservers=ppservers)
def mpsum(array):
      pool = multiprocessing.Pool(2);
      half = len(array)/2
      s = sum(pool.map(sum, [array[:half], array[half:]]));
      return s
arrays = [np.ones(5000) for i in xrange(10) ]
imports = ("multiprocessing",)
depfuncs = tuple()
jobs = [serv.submit(mpsum,(a,), depfuncs, imports) for a in arrays];
s = sum([job() for job in jobs]);
print s
```

Рис. 2. Сценарий для параллельного суммирования массивов средствами Multiprocessing и ParallelPython




```
import numpy as np
from scipy import stats
from mayavi import mlab
mu, sigma = 0, 0.5
x,y,z = [10*np.random.normal(mu, sigma, 100) for i in [1,2,3]]
kde = stats.gaussian_kde(np.vstack([x,y,z]))
xmin, ymin, zmin = x.min(), y.min(), z.min()
xmax, ymax, zmax = x.max(), y.max(), z.max()
xi, yi, zi = np.mgrid[xmin:xmax:30j, ymin:ymax:30j, zmin:zmax:30j]
coords = np.vstack([item.ravel() for item in [xi, yi, zi]])
density = kde(coords).reshape(xi.shape)
figure = mlab.figure('DensityPlot')
grid = mlab.pipeline.scalar_field(xi, yi, zi, density)
min, max = density.min(), density.max()
mlab.pipeline.volume(grid, vmin=min, vmax=min + .5*(max-min))
mlab.points3d(x, y, z, scale_factor=1)
mlab.axes()
mlab.show()
```

Рис. 3. Сценарий для расчета плотности распределения точек и ее трехмерной визуализации с использованием Mayvi2 и SciPy

в большинстве случаев доступ к данным с узла, которому требуется прочитать и обработать их, открыт только по SSH. Для решения этой задачи подходит библиотека Paramiko.

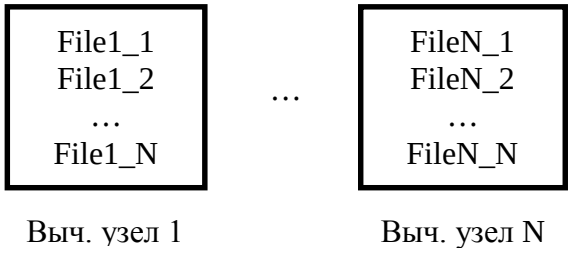


Рис. 4. Схема расположения данных на вычислительных узлах

Paramiko [9] – библиотека для языка Python, предоставляющая реализацию и интерфейс для взаимодействия с удаленными системами по протоколу SSHv2. В данной библиотеке имеется как реализация клиента, так и реализация сервера. Кроме этого, Paramiko предоставляет удобный API, реализующий объекты типа file, представляющие собой файлы на удаленной системе. Данная функциональность легла в основу реализации SSH сборщика в представленной системе.

РЕАЛИЗАЦИЯ

Используя приведенные выше средства, была начата разработка системы, позволяющей достичь намеченных целей, а именно параллельного чтения и обработки данных, а также их визуализации.

В качестве начального этапа был написан пакет mmdlab, реализующий API общего назначения для таких задач. Ниже будут описаны возникшие в процессе реализации проблемы, прикладные задачи и способы их решения с использованием разрабатываемой библиотеки. Также будет обращено внимание на особенности реализации некоторых участков пакета.

Получение данных. Модуль для чтения и частичной обработки данных получил название datareader. В этом модуле были реализованы необходимые объекты для чтения и хранения данных, такие как Container, Parser и средства доступа к файлам с локальной файловой системы и по SSH. В терминологии модуля mmdlab, Container – это структура, хранящая в себе прочитанные данные в определенном пользователем формате. Parser – это специальный объект, который читает бинарные данные и разбирает их структуру, в результате получая контейнер. Сами исходные данные Parser получает от объекта класса Transport, который представляет собой интерфейс доступа к файловой системе, локальной или удаленной. Наследуя и комбинируя объекты от данных классов, пользователь может легко использовать данные любого формата и с доступом по любому протоколу, например SSH или HTTP.

Рассмотрим процедуру чтения одного конкретного состояния МД системы из работы [1].

Учитывая распределенную структуру входных данных, единичное состояние системы представляет собой множество файлов с атомистическими данными, каждый из которых нужно прочитать, разобрать бинарную структуру и собрать в один общий контейнер, хранящий в себе всю картину симулируемой системы.

Для ускорения чтения и обработки требуется использовать параллельный алгоритм: мастер-процесс запускает N дочерних процессов-загрузчиков и по очереди начинает отдавать файлы каждому освободившемуся процессу. Когда процесс-загрузчик закончил чтение и сборку своей части контейнера, мастер-процесс объединяет прочитанные данные с мастер-контейнером, а затем назначает процессу-загрузчику новый файл. После того, как все процессы-загрузчики завершились и больше нет файлов для чтения, мастер-процесс проводит необходимую обработку контейнера, где уже находятся все доступные данные, и отдает его по цепочке следующему на очереди обработчику.

Для того чтобы расширить возможности модуля mmdlab на чтение пользовательских данных, потребуется описать новые сущности для хранения и загрузки таких данных. В качестве примера можно рассмотреть определение таких объектов для чтения формата CSV (Comma-Separated Values) с тремя колонками. На рис. 5 приведен пример такого расширения для чтения CSV с разных узлов по протоколу SSH. На практике пользователю потребуется описать новый класс, наследуемый от класса DummyContainer и переопределить в нем метод append_data. Также потребуется описать свой класс для разбора данных. Рассмотрим некоторые особенности реализации параллельного загрузчика.

```
class CsvContainer(dr.containers.DummyContainer):
      def __init__(self):
            self.cols = [[],[],[]]
      def append_data(self,data):
            for i,d in enumerate(data[:]):
                  self.cols[i].extend(d)
class CsvParser(dr.parsers.DummyParser):
    def data(self):
        cols = [[],[],[]]
        for line in self.transport.readlines():
            c = line.split(",")
            for i in range(0,3):
                 cols[i].append(c[i])
        return cols
nodes = ({"ip":"10.0.0.1","pwd":"12345","login":"test"},
             {"ip":"10.0.0.2","pwd":"12345","login":"test"})
remote_dirs = [(sys.argv[1], node) for node in nodes]
transport = dr.transport.RemoteDirs(remote_dirs)
parser = CsvParser()
reader = dr.DistributedDataReader(file_mask="1*.csv",
                                  transport=transport,
                                  parser = parser,
                                  container = CsvContainer)
container = mmdlab.run([reader, ])
```

Рис. 5. Сценарий для чтения пользовательского формата данных со множества узлов

Проблема утечки оперативной памяти в режиме многопоточной обработки. Обратим внимание на функцию чтения из класса DistributedDataReader (рис. 6). При реализации данного метода была выявлена проблема утечки памяти. После запуска пула процессов и выполнения множества задач в нем, потребление памяти резко возрастало. Оказалось, что по умолчанию дочерние процессы интерпретатора, созданные библиотекой Multiprocessing, обрабатывают все запланированные задачи, не перезапускаясь. Каждая выполняемая в таких процессах задача оставляет свой контекст, тем больший по размеру потребляемой памяти, чем больше данных возвращает задача.

В результате после длительной работы программы у вычислительного узла заканчивалась оперативная память. Предлагаемое нами решение этой проблемы состоит в следующем. У объекта пула процессов имеется специальный параметр конструктора maxtaskperchild, позволяющий задать количество задач на один процесс, после чего управляющий алгоритм его перезапустит. Изменение этого параметра позволяет варьировать максимальный объем потребляемой памяти. Однако стоит заметить, что чем меньше этот параметр, тем чаще мастер-процесс будет перезапускать дочерние процессы интерпретаторов, что может занимать существенное время. В рамках рассматриваемой задачи это время не является критичным, и установка достаточно маленького значения вполне оправданна.

На рис. 7 представлена зависимость времени чтения от значения параметра maxtaskperchild для 256 файлов и пула из одного потока.

Как видно из графика, оптимальным поведением пула процессов является перезапуск процессов-рабочих каждые 16 заданий. Это позволяет ограничить потребление оперативной памяти, при этом оставляет влияние времени перезапуска процесса-рабочего на общее время чтения почти незаметным.

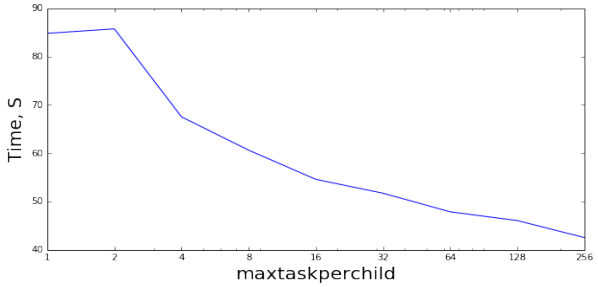


Рис. 7. Зависимость времени параллельного чтения от количества задач на процесс

MultiPool. Другая проблема, возникшая в процессе разработки, заключалась в том, что по умолчанию библиотека multiprocessing не позволяет создавать "вложенные" пулы процессов. В частности, если у пользователя возникнет необходимость запустить параллельный процесс чтения множества состояний исследуемой системы, к примеру, для построения траекторий частиц, то Multiprocessing не позволит сделать это. В решении этой проблемы помогает интроспекция, которая поддерживается языком Python в полной мере. В разработанный в данной работе модуль mmdlab была включена конструкция, показанная на рис. 8. Она подменяет методы _get_daemon и _set_daemon у класса Process и предоставляет новый объект, наследованный от класса Pool. Ее нужно использовать вместо стандартного класса Pool из модуля Multiprocessing.

Обработка данных. Для обработки и фильтрации данных в разработанной библиотеке mmdlab используются те же механизмы, что и чтение данных. Используется так называемая «конвейерная» архитектура, когда объект контейнера с данными передается по цепочке через множество обработчиков, которые могут его изменять, дополнять или создавать новые

```
class DistributedDataReader:
    ...
    def read(self):
        ...
        files = self.transport.list(self.file_mask)
        container = self.container()
        pool = Pool(processes=self.np, maxtasksperchild=self.mtpc)
        results = [pool.apply_async( rd,
                                args=(f,self.transport.filer(),self.parser))
                                for f in files]
        for ct in results:
            container.append_data(ct.get())
            return container.finalize()
```

Рис. 6. Вход в процедуру чтения данных в классе DistributedDataReader

```
import multiprocessing
import multiprocessing.pool
class NoDaemonProcess(multiprocessing.Process):
    def _get_daemon(self):
        return False
    def _set_daemon(self, value):
        pass
    daemon = property(_get_daemon, _set_daemon)
class MultiPool(multiprocessing.pool.Pool):
    Process = NoDaemonProcess
```

Рис. 8. Подмена методов у класса Pool для возможности запуска процессов внутри процессов

контейнеры. Главный обработчик библиотеки передает контейнер, полученный от предыдущего задания, на вход методу обработки. Реализация этих методов может быть как последовательная, так и параллельная.

В приложении к конкретной задаче анализа результатов молекулярно-динамического моделирования из работы [1], в mmdlab были добавлены объекты для постобработки данных.

Например, фильтрация частиц по различным критериям, в частности для отсечения частиц вне заданной области, для фильтрации по индексам и разделения частиц по типам. Все вычислительно емкие процедуры были оптимизированы с использованием Numpy+Numba.

В качестве примера рассмотрим задачу визуализации положения и температуры частиц, разделенных по типу, в заданной области. С использованием библиотеки mmdlab такая задача решается следующим образом.

Пользователь создает объект загрузчика данных, с указанием их местоположения и временной метки, указывает описание частиц, используя которое, будет работать фильтр разделения, и создает соответствующие объекты фильтров (фильтр частиц по положению в области, фильтр разделения). После этого передает эти объекты в конвейер. Расчет температуры производится при создании контейнера на одном из этапов выполнения. На рис. 9 показан листинг такой процедуры и результат выполнения.

Визуализация. Для визуализации в данной работе использовались библиотеки Mayavi2 и Matplotlib. Для удобства использования часто применяемых способов отрисовки в состав mmdlab был включен модуль vis, который представляет собой обертку над методами этих библиотек, комбинирующий их возможности для достижения заданного результата.

Ввиду однопоточной архитектуры Mayavi и Matplotlib, процесс визуализации данных на данный момент поддерживается только в однопоточном режиме в рамках одного процесса. Однако mmdlab позволяет запустить гибридные задачи чтения и отрисовки на множестве узлов и в многопроцессном режиме, что значительно ускоряет рендеринг видеоанимации.

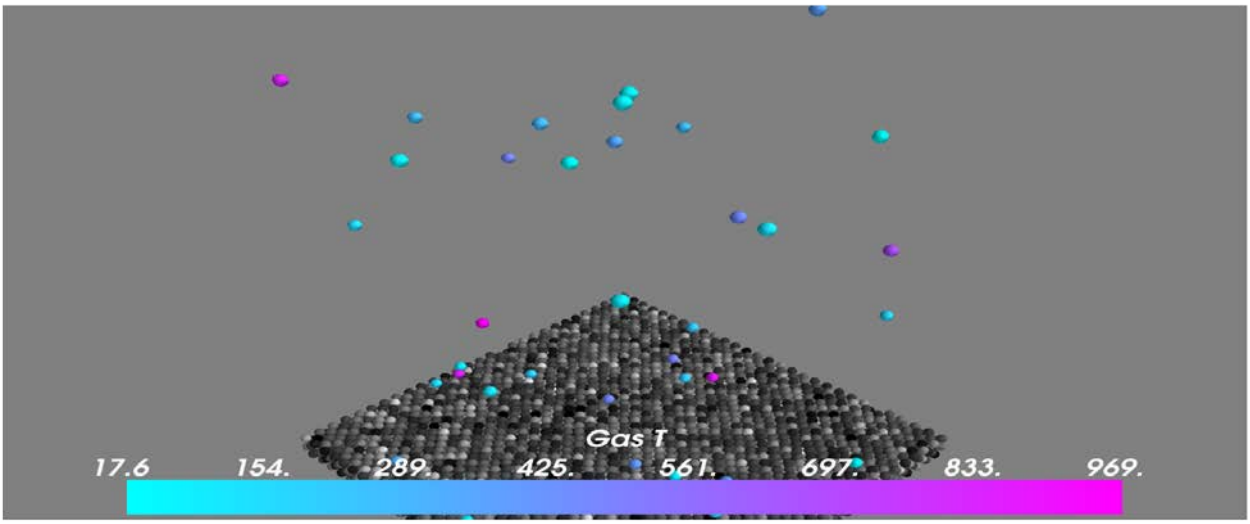
Например, рассмотрим задачу составления анимации, состоящей из кадров, представляющих собой состояния исследуемой системы частиц в последовательные моменты времени. Исходные данные распределены по множеству узлов, следовательно, визуализацию можно запустить на каждом из узлов и потом собрать результаты. Для реализации такой задачи предлагается следующий алгоритм:

1) на каждом из заданных узлов запустить последовательность чтения и визуализации;
2) собрать все отрисованные кадры на мастер-узле;
3) собрать из этих кадров gif-анимацию.

Для сборки анимационного файла GIF формата используется программа convert из пакета ImageMagick [10]. Для сборки результатов отрисовки в видеофайл mmdlab использует ffmpeg.

СИСТЕМА УПРАВЛЕНИЯ ЗАДАЧАМИ

В данной работе было рассмотрено веб-приложение IPython notebook, позволяющее выполнять пользовательские Python скрипты, используя веб-браузер. Учитывая то, что IPython позволяет выполнять скрипты поблочно, с сохранением контекста предыдущей задачи, использование такого средства оказалось удобным для предварительной обработки и просмотра результатов. Используя его, возможно, например, вызывать процедуру чтения для одного состояния исследуемой системы, поэкспериментировать с выбором региона отсечения




```
from mmdlab.datareader.shortcuts import read_distr_gimm_data
import mmdlab
import sys
reader = read_distr_gimm_data(sys.argv[1],sys.argv[2])
filter_reg = mmdlab.dataprocessor.filters.RegionFilter([0,10,0,10,0,10])
parts_descr = \
{ "Nickel" : { "id" : 0, "atom_mass" : 97.474, "atom_d" : 0.248}, \
"Nitrogen" : { "id" : 1, "atom_mass" : 46.517,"atom_d" : 0.296} }
filter_split = mmdlab.dataprocessor.filters.SplitFilter(parts_descr)
container = mmdlab.run([reader,  filter_reg, filter_split ])
met,gas = container["Nickel"], container["Nitrogen"]
mp = mmdlab.vis.Points3d(met, scalar=met.t, size=met.d, colormap="black-white")
gp = mmdlab.vis.Points3d(gas, scalar=gas.t, size=gas.d, colormap="cool")
mmdlab.vis.colorbar(gp, "Gas T")
mmdlab.vis.show(distance=20)
```

Рис. 9. Сценарий для визуализации частиц и их температур, разделенных по типу, в заданной области

частиц, подобрать нужное положение камеры над сценой и запустить задачу визуализации всех состояний с подобранными параметрами.

ЗАКЛЮЧЕНИЕ

В данной работе представлена пробная версия высокоуровневой библиотеки для языка Python, позволяющая легко кластеризовать и распараллелить задачи чтения, обработки и визуализации различных данных. Основными задачами при разработке данной библиотеки были анализ и визуализация данных, полученных в результате молекулярно-динамического моделирования эволюции микросистемы газ-металл [1]. Использование библиотеки позволило в деталях рассмотреть эффект адсорбции азота на никелевой пластине (рис. 10), включая анализ траекторий отдельных частиц. Особое внимание было уделено возможности расширения созданной библиотеки, которая реализована благодаря гибкости использованных инструментов. В результате использование разработанной библиотеки можно распространить на чтение и визуализацию практически любых структур данных.

СПИСОК ЛИТЕРАТУРЫ

1. **Подрыга В. О., Поляков С. В., Пузырьков Д. В.** Суперкомпьютерное молекулярное моделирование термодинамического равновесия в микросистемах газ-металл // Вычислительные методы и программирование. 2015. Т. 16, № 1. С. 123–138. [V. O. Podryga, S. V. Polyakov and D. V. Puzyrkov, “Supercomputer Molecular Modeling of Thermodynamic Equilibrium in Gas–Metal Microsystems” (in Russian), in *Vychislitel'nye Metody i Programmirovanie*, vol. 16, no. 1, pp. 123-138, 2015.]

2. **Официальная документация Python** [Электронный ресурс]. URL: https://www.python.org/ (дата обращения 22.11.2015). [(2015, Nov. 22). Python official documentation [Online]. Available: https://www.python.org/]

3. **Fernando Pérez, Brian E. Granger.** IPython: A System for Interactive Scientific Computing // Computing in Science and Engineering. 2007. Vol. 9, No. 3. P. 21–29. [Электронный ресурс]. URL: http://ipython.org (дата обращения 04.02.2015). [P. Fernando, E. G. Brian, IPython: A System for Interactive Scientific Computing (2015, Feb, 4) (in English), in

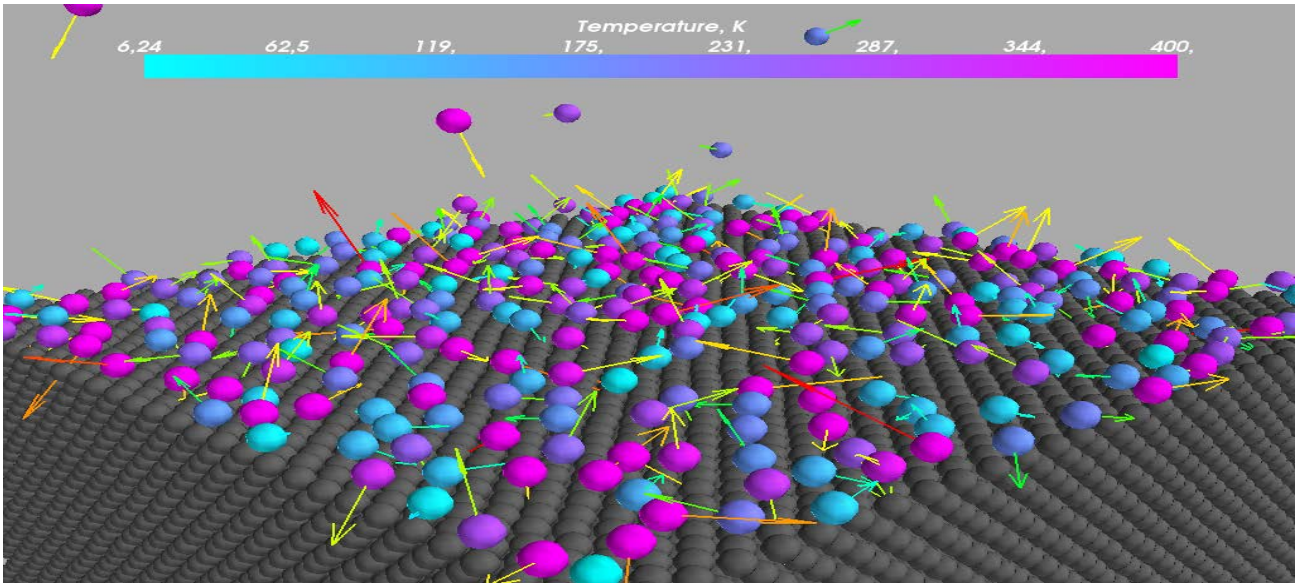


Рис. 10. Результат визуализации эффекта адсорбции на выбранном участке исследуемой системы

Computing in Science and Engineering, vol. 9, no. 3, pp. 21–29, 2007, [Online]. Available: http://ipython.org]

4. **Официальная документация Numpy** [Электронный ресурс]. URL: http://www.numpy.org/ (дата обращения: 04.02.2016) [(04, Feb. 2016). Numpy official documentation [Online]. Available: http://www.numpy.org/]

5. **Официальная документация Numba** [Электронный ресурс]. URL: http://www.numba.pydata.org/ (дата обращения: 04.02.2016) [(04, Feb. 2016). Numba official documentation [Online]. Available: http://www.numba.pydata.org/]

6. **Официальная документация ParallelPython** [Электронный ресурс]. URL: http://www.parallelpython.com/ (дата обращения: 04.02.2016) [(04, Feb. 2016). ParallelPython official documentation [Online]. Available: http://www.parallelpython.com/]

7. **Официальная документация Mayavi2** [Электронный ресурс]. URL: http://docs.enthought.com/mayavi (дата обращения: 04.02.2016) [(04, Feb. 2016). Mayavi2 official documentation [Online]. Available: http://docs.enthought.com/mayavi]

8. **Официальная документация Matplotlib** [Электронный ресурс]. URL: http://matplotlib.org/ (дата обращения: 04.02.2016) [(04, Feb. 2016). Matplotlib official documentation [Online]. Available: http://matplotlib.org/]

9. **Официальная документация Paramiko** [Электронный ресурс]. URL: http://www.paramiko.org/ (дата обращения: 04.02.2016) [(04, Feb. 2016). Paramiko official documentation [Online]. Available: http://www.paramiko.org/]

10. **Официальная документация ImageMagick** [Электронный ресурс]. URL: http://www.imagemagick.org/ (дата обращения: 04.02.2016) [(04, Feb. 2016). ImageMagick official documentation [Online]. Available: http://www.imagemagick.org/]

ОБ АВТОРАХ

ПУЗЫРЬКОВ Дмитрий Валерьевич, аспирант ИПМ им. М. В. Келдыша РАН. Дипл. инженер-математик (МИЭТ, 2014). Готовит дис. по программному обеспечению суперкомпьютерных экспериментов в обл. нанотехнологий.

ПОДРЫГА Виктория Олеговна, ст. науч. сотр. ИПМ им. М. В. Келдыша РАН. Дипл. математик, сист. программист (МГУ имени М. В. Ломоносова, 2007). Канд. физ.-мат. наук по мат. моделированию (МГУ имени М. В. Ломоносова, 2011). Иссл. в обл. мат. моделирования в нанотехнологиях.

ПОЛЯКОВ Сергей Владимирович, зав. сектором ИПМ им. М. В. Келдыша РАН, проф. НИЯУ МИФИ. Дипл. математик (МГУ имени М. В. Ломоносова, 1987). Д-р физ.-мат. наук по мат. моделированию (ИПМ им. М. В. Келдыша РАН, 2011). Иссл. в обл. мат. моделирования в нанотехнологиях.

METADATA

Title: Distributed data processing in application to the molecular dynamics simulation of equilibrium state in the gas-metal microsystems.

Authors: D.V. Puzyrkov[1], V.O. Podryga[2], S.V. Polyakov[3,4]

Affiliation:

[1,2,3] Keldysh Institute of Applied Mathematics (Russian Academy of Sciences)

[4] National Research Nuclear University MEPhI

Email: [1] dpuzyrkov@gmail.ru, [2] pvictoria@list.ru, [3,4] polyakov@imamod.ru.

Language: Russian.

Source: Vestnik UGATU (scientific journal of Ufa State Aviation Technical University), vol. 20, no. 1 (71), pp. 175–186, 2016. ISSN 2225-2789 (Online), ISSN 1992-6502 (Print).

Abstract: In this paper a software package for distributed processing and visualization of the molecular dynamics simulation is presented. During the creation process the performance issues of the Python programming language were studied, as well as the abilities to avoid them. It was also studied the problem of obtaining and processing distributed data stored on multiple computational nodes. An open-source package for scientific visualization Mayavi2 was chosen as a visualization tool. As a result of application this system to the data obtained from MD simulation of the gas and metal plate interaction we were able to observe in the details the effect of adsorption, which is important for many practical applications.

Key words: Distributed data processing; visualization; molecular dynamics; cloud computing; parallel algorithms; Python; Mayavi2; Numba.

About authors:

PUZYRKOV, Dmitry Valerievich, Postgrad. (PhD) Student, KIAM (RAS). Master of Applied Mathematics (MIET, 2014).

PODRYGA, Viktoriia Olegovna, Senior Researcher, KIAM (RAS). Dipl. Mathematician, System Programmer (Lomonosov Moscow State Univ., 2007). Cand. of Phys.-Math. Sci. (Lomonosov Moscow State Univ., 2011).

POLYAKOV, Sergey Vladimirovich, Head of Sector, KIAM (RAS); Prof., MEPhI. Dipl. Mathematician (Lomonosov Moscow State Univ., 1987). Cand. of Phys.-Math. Sci. (KIAM (RAS), 1992), Dr. of Phys.-Math. Sci. (KIAM (RAS), 2011).