Распределенная визуализация в задачах моделирования термодинамического равновесия в микросистемах газ-металл методами молекулярной динамики*

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

Федеральное государственное бюджетное учреждение науки Институт прикладной математики им. М.В. Келдыша Российской академии наук[1]

В данной работе представлен программный комплекс для обработки и визуализации результатов моделирования задач молекулярной динамики. В процессе разработки комплекса были изучены вопросы производительности интерпретируемого языка Python и средства, позволяющие ее увеличить. Так же 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: визуализация, молекулярная динамика, облачные вычисления, параллельные алгоритмы, Python, Mayavi2, Numba

1. 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. Математические модели такого типа могут оперировать огромным количеством частиц: от миллионов до миллиардов. При этом каждая частица может описываться десятками параметров. Объемы выходных данных в подобных ВЭ исчисляются терабайтами информации.

В настоящее время наиболее используемым подходом для ускорения масштабных вычислений является их распараллеливание, когда множество вычислительных узлов занимается обработкой только небольшой части задачи, что существенно уменьшает время, нужное для полной симуляции системы. Однако такой способ ускорения вычислений ведет к нескольким проблемам, связанным с хранением результатов. Чаще всего после произведения вычислений вычислительные узлы обмениваются данными, и мастер-процесс составляет полную систему в памяти и сохраняет ее одним большим файлом.

Что делать в случае, если размер всей системы превышает оперативную память мастер-узла? В таком случае каждый вычислительный узел хранит свои результаты обособленно. Такой способ хранения дает несколько преимуществ. Первое - отсутствие необходимости последовательного чтения всех результатов для последующей обработки. Каждый вычислительный узел читает только свои данные. Второе - каждый отдельный файл данных обычно не очень большой, и следовательно его обработка будет занимать меньше времени. Расчетные программы такого типа и рассматриваются в данной статье, а именно алгоритм и его реализация, описанные в [1].

Одним из способов представления данных ВЭ является двух- и трехмерная визуали-

*Работа выполнена при поддержке РФФИ (гранты № 15-07-06082-а, № 15-29-07090-офи_м)

зация. 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 данные 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 любым программным обеспечением для научной визуализации. Проблемы такого способа хранения и отрисовки в ограниченности возможностей собственного программного обеспечения расчетного комплекса в отношении визуализации, а в случае общеизвестных стандартов это проблема больших файлов.

В данной работе представлена попытка создать программный комплекс, позволяющий в интерактивном режиме импортировать, обрабатывать 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 работы вычислительной программы, описанной в [1], которые в виду параллельности алгоритма и особенностей способа хранения представляют собой как один большой файл, описывающий состояние системы, так и распределенные по вычислительным узлам файлы. Данные, полученные в результате моделирования, представляют собой информацию взаимодействия молекул газа с атомами металла в технических микросистемах, реализующих различные нанотехнологии.

2. Постановка задачи

Задача сбора и обработки распределенных данных, полученных в результате выполнения некоторой расчетной программы, имеет несколько ключевых особенностей. Во-первых, это специфичность задачи. В результате исследования не было найдено подходящих средств для параллельной загрузки распределенных однотипных данных, что послужило поводом для данного исследования. Во-вторых это размер входных данных. Они могут быть как небольшим одномерным массивом, так и большим количеством файлов, разбросанных по разным узлам и директориям. Такие задачи обычно решаются либо средствами этого программного комплекса, либо разработкой специализированного обработчика, который умеет понимать формат вывода используемой расчетной программы. В третьих, это необходимость обрабатывать такие результаты для удобного представления на графиках или в 3D визуализации. В связи с этим 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 понимать, что в данном случае большую роль играет легкая расширяемость такой системы, чтобы ее можно было легко использовать для обработки данных, хранящихся в любом формате.

В качестве начального этапа разработки была выбрана задача визуализации данных, полученных из [1]. Эта задача подразумевает использование всех особенностей системы, так как результаты вычислений разбросаны по вычислительным узлам с доступом по SSH и требуют постобработки.

3. Средства разработки

3.1. Python

Python [2] - интерпретируемый язык программирования. Он широко используется в научном сообществе. Его дизайн настаивает на написании хорошо читаемого кода, а его синтаксис позволяет описывать алгоритмы за меньшее число строк, чем это возможно на C++ или Java. Он кросплатформенный. Все это очень важно при разработке больших приложений. Синтаксис ядра Python очень прост и краток, в то же время стандартная библиотека дает большой объем полезных функций и удобные структуры данных. Python поддерживает несколько парадигм программирования, в том числе объектно-ориентированную, императивную и функциональную. Так же он имеет динамическую "утиную" типизацию и автоматическое управление памятью. Хотя Python уже имеет версию 3, в данной работе использовалась версия Python 2.7, в виду того, что огромное количество библиотек необходимых было написано на Python 2.7, а Python 3 и Python 2.7 в некоторых случаях обратно не совместимы.

3.2. IPython

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, BASH прямо в браузере, а так же взаимодействовать с объектами визуализации. В данной работе это используется как основная командная оболочка для управления системой.

3.3. Ускорители вычислений

Несмотря на все достоинства, у основной реализации интерпретатора, CPython [2], имеется достаточно большой минус, связанный со скоростью выполнения и многопоточностью. Связано это с 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.

3.3.1. Numpy

Numpy [4] - библиотека с открытым кодом, представляющая бесплатную, но сильную альтернативу проприетарным решениям для матричных вычислений, таких как MATLAB. Этот пакет содержит в себе множество алгоритмов для анализа данных, реализованных на низком уровне. Это означает их независимость от GIL, а следовательно их использование существенно увеличивает производительность по сравнению с кодом на Python. Имеется очень подробная документация. Все эти особенности вместе с подробной документацией делают Numpy идеальным выбором для ускорения вычислений на языке Python.

3.3.2. Numba

Numba [5] представляет собой оптимизирующий Just-In-Time (JIT) компилятор, позволяющий заметно ускорить критические к времени выполнения участки кода, используя простейшие средства языка, такие как декораторы. Это открытое программное обеспечение (ПО), построенное над инфраструктурой LLVM(Low Level Virtual Machine). Благодаря возможностям аннотации типов, отключения GIL и выхода за рамки объектов Python, ком-

пилятор Numba может генерировать более эффективный и оптимизированный байткод, не меняя при этом саму Python основу напрямую. Более того, Numba старается автоматически векторизовать то, что может, используя возможности многопроцессорных систем по максимуму. Для примера: Intel Core I7, перемножение массивов с умножением и делением на константу (Рис. 1, Таблица 1).

Таблица 1. Сравнение производительности

N	Numpy	Numba+Numpy	Отн. прирост производительности
1000000	0.19 мс	0.07 мс	2.77
10000000	1.62 мс	0.74 мс	2.19
100000000	16.06 мс	7.4 мс	2.17

```
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 перемножения массивов средствами Numpy и Numba

Как видно из таблицы и из листинга, Numba позволяет ускорить вычислительную часть кода почти в два раза, не прибегая ни к каким особым оптимизациям, как бы скорее всего пришлось делать при использовании любого другого средства. Естественно, что Cython [6], в виду того, что он транслирует Python код в язык C, а потом его компилирует в машинные коды (Numba для этого использует LLVM), даст в данном случае большую производительность, однако это также потребует явного определения типов и написания кода на диалекте Cython, а не на чистом Python, и, возможно, потребует дополнительных оптимизаций.

3.4. Средства параллелизации

В виду наличия у CPython такого механизма как GIL, параллелизация приложений с помощью потоков не возможна. 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 многопроцессорное приложение на небольших данных не рационально. Существует несколько инструментов для удобного управления такими задачами.

3.4.1. Multiprocessing

Multiprocessing [7] - модуль стандартной библиотеки, предоставляющий интерфейс для взаимодействия с процессами сходный с интерфейсом управлением потоками. Кроме этого, он добавляет некоторый новый функционал, например класс Pool, представляющий собой абстракцию и управляющий механизм для набора параллельных процессов интерпретатора. Он так же реализует межпроцессорные примитивы, такие как очередь и mutex.

3.4.2. ParallelPython

ParallelPython [8] - библиотека для решения проблемы кластеризации приложения. Ее реализация имеет клиент-серверную структуру и требует установку серверной части на вычислительные узлы. После того, как эта процедура будет завершена, ParallelPython позволяет практически в одну строку запустить выполнение задач на множестве узлов, объединенных сетью, то есть в кластерном режиме. Данная библиотека сама балансирует нагрузку и перезапускает задачи на другом узле, если исполнитель перестал отвечать. Вместе с модулем Multiprocessing он 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 2 процесса с помощью ParallelPython и каждый из них запускает еще 2 процесса средствами Multiprocessing.

```
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 средствами ParallelPython и Multiprocessing

3.5. Визуализация

Существует множество сторонних средств для визуализации данных. Для примера можно рассмотреть Paraview, VMD, Aviso, Tecplot. Каждая из перечисленных программ имеет собственный формат хранения данных, а так же умеет читать стандартизованные форматы. Представленную в данной работе библиотеку можно использовать как средство для подготовки данных для последующей визуализации в перечисленных программах, однако также она имеет встроенные средства визуализации, практически не уступающие по возможностям перечисленным программам. В качестве пакетов для визуализации общего назначения были выбраны следующие.

3.5.1. Mayavi2

Mayavi2 [9] - программное обеспечение для визуализации общего назначения. Оно предоставляет пользователю возможности для загрузки и отрисовки данных во встроенном приложении, а так же удобный Python интерфейс для программной генерации изображений. Данная библиотека построена поверх известной в научных кругах библиотеки VTK, что позволяет интегрировать ее с программами, поддерживающими формат данных VTK. Mayavi2 предоставляет широкие возможности для визуализации данных, начиная от гидродинамических расчетов и заканчивая атомистическими данными. В случае интерактивного использования так же доступны инструменты для изменения параметров визуализации на лету, таких как размеры объектов, цветовые схемы, параметры фильтров, добавление на

сцену текста и заголовков. Так же, Mayavi2 имеет возможность offscreen-отрисовки (в оперативной памяти), что чрезвычайно важно для серверной и пакетной обработки большого количества данных. На Рис. 3 и Рис. 4 представлен пример расчета плотности распределения точек и ее трехмерная визуализация с использованием Mayavi2 и библиотеки для научных вычислений SciPy.

```
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

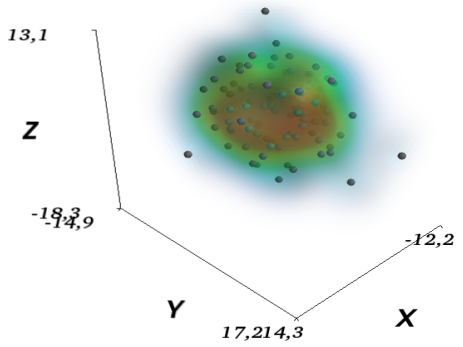

Рис. 4. Результат визуализации плотности распределения точек.

3.5.2. Matplotlib

Matplotlib [10] — это Python-библиотека для построения качественных двухмерных графиков. Она широко используется в научном сообществе. Использование Maplotlib 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.

3.6. Другие средства

3.6.1. Paramiko

Paramiko [11] - 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. Также Paramiko предоставляет удобные объекты, реализующие объекты типа file, представляющие собой файлы на удаленной системе. Данная функциональность легла в основу реализации SSH сборщика в представленной системе. На Рис. 5 показан простейший пример получения списка файлов из заданной директории и чтения первого из них.

```
import paramiko
client = paramiko.SSHClient()
client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
client.connect(address,username=login, password=passwd)
stdin, stdout, stderr = client.exec_command("ls␣./")
files = [f.strip() for f in  stdout.readlines()]
t = paramiko.Transport((self.server, 22))
t.connect(username=login, password=passwd)
sftp = paramiko.SFTPClient.from_transport(t)
f = sftp.file(files[0],"r")
print f.read(1024)
```

Рис. 5. Сценарий для чтения первого файла из удаленной директории по протоколу SSH, используя Paramiko

4. Реализация

Используя приведенные в разделе "Средства разработки" инструменты, была начата разработка системы, которая позволит достичь намеченных целей, а именно параллельного чтения и обработки данных, а так же их визуализацию. В качестве начального этапа, был написан пакет mmdlab, реализующий все эти возможности. Ниже будут описаны задачи и способы их решения с использованием разрабатываемой библиотеки. Также будет обращено дополнительно внимание на особенности реализации некоторых ее участков.

4.1. Получение данных

Модуль для чтения и частичной обработки данных получил название datareader. Возможности, которые предоставляет этот модуль, заключаются в реализации параллельных механизмов чтения данных, с удобным для пользователя интерфейсом и возможностями расширения. Рассмотрим процедуру чтения одного конкретного состояния системы. Учитывая распределенную структуру входных данных, единичное состояние системы может быть представлено как одним большим массивом данных, так и множеством файлов, находящихся на различных вычислительных узлах или в локальной директории. В таком случае каждый процесс чтения разбивается на несколько процессов, загружающих в память каждый свою порцию данных. После того как все дочерние процессы прочитали и, возможно, обработали данные, мастер-процесс собирает их воедино и передает следующему обработчику по цепочке.

На Рис. 6 приведен пример сценария чтения распределенных данных из локальной директории.

Для того чтобы пользователь мог читать данные своего заданного формата, ему потребуется реализовать свои объекты для их хранения и загрузки. В качестве примера можно рассмотреть определение таких сущностей для чтения формата CSV (Comma-Separated Values) с тремя колонками. Загрузка таких данных, хранящихся на множестве узлов, будет

```
import mmdlab
from mmdlab import datareader as dr
import sys
transport = dr.transport.LocalDir(sys.argv[1])
parser = dr.parsers.GimmBinaryParser()
reader = dr.DistributedDataReader(file_mask="state0_*.dat",
                                  transport=transport,
                                  parser = parser)
container = mmdlab.run([reader, ])
```

Рис. 6. Сценарий для чтения распределенных данных из локальной директории.

выглядеть как показано на Рис. 7.

```
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. 7. Сценарий для чтения пользовательского формата данных со множества узлов.

Рассмотрим некоторые особенности реализации параллельного загрузчика. Обратим внимание на функцию чтения из класса DistributedDataReader (Рис. 8).

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 процесс интерпретатора, созданный библиотекой mutithreading, по умолчанию обрабатывает все запланированные задачи, не перезапускаясь. Каждая выполняемая в нем задача оставляет свой контекст, тем больший по размеру потребляемой памяти, чем больше возвращаемых им данных. В результате после длительной работы программы у вычислительного узла заканчивалась оперативная память. Однако решение оказалось простым - у объекта пула процессов имеется специальный параметр конструктора maxtasksperchild, позволяющий задать количество задач на один процесс, после чего управляющий алгоритм его перезапустит. Изменение этого параметра позволяет варьировать максимальный объем потребляемой памяти. В рамках рассматриваемой задачи, время, которое занимает пересоздание процесса, не существенно, и такой подход вполне оправдывает себя.

Еще одна проблема, с которой пришлось столкнуться, - невозможность создавать "вложенные" пулы процессов. Например, если пользователь захочет запустить процесс чтения множества состояний системы так же в параллельном режиме, как и процесс чтения одного состояния, используя пул процессов (Рис. 9), он будет огорчен: по умолчанию библиотека

```
class DistributedDataReader:
    ...
    def read(self, np=32):
        files = self.transport.list(self.file_mask)
        container = self.container()
        pool = Pool(processes=np, maxtasksperchild=16)
        results = [pool.apply_async( rd,
                                args=(f, self.transport, self.parser))
                                for f in files]
        for ct in results:
            container.append_data(ct.get())
                return container.finalize()
```

Рис. 8. Вход в процедуру чтения данных в классе DistributedDataReader

multiprocessing не позволит сделать это.

```
def read_state(mask):
        import mmdlab
        from mmdlab import datareader as dr
        import sys

        transport = dr.transport.LocalDir(sys.argv[1])
        parser = dr.parsers.GimmBinaryParser()
        reader = dr.DistributedDataReader(file_mask=mask,
                            transport=transport,
                            parser = parser)
        return  mmdlab.run([reader, ])

import multiprocessing as mp
results = Pool(2).map(read_state, ["state0[*].dat", "state1[*].dat"])
```

Рис. 9. Сценарий, использующий пул процессов для чтения множества контрольных точек

В решении этой проблемы помогает интроспекция, которая поддерживается языком в полной мере. Разработанный в данной работе модуль включает в себя конструкцию (Рис. 10), подменяющую методы _get_daemon и _set_daemon у класса Process и предоставляющую новый объект, наследованный от класса Pool, позволяющую выполнить код на Рис. 10. Ее нужно использовать вместо стандартного класса Pool из модуля Multiprocessing.

```
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. 10. Пул процессов для выполния чтения множества контрольных точек

4.2. Обработка данных

Обработка данных в разработанной библиотеке использует те же механизмы, что и чтение данных. 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 МД моделирования были разработаны методы пост-обработки данных, например, филь-

трация частиц, в частности для отсечения частиц вне заданной области, для фильтрации по индексам и разделении 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 реализованной библиотеки такая задача решается как показано на Рис. 11. Результат визуализации показан на Рис. 12.

```
from mmdlab.datareader.shortcuts import read_distr_gimm_data
import mmdlab
import sys
reader = read_distr_gimm_data(sys.argv[1],"*.dat")
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)
temp_proc = mmdlab.dataprocessor.Temperature()
container = mmdlab.run([reader,  filter_reg, filter_split, temp_proc ])
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()
```

Рис. 11. Сценарий для визуализации частиц и их температур, разделенных по типу, в заданной области

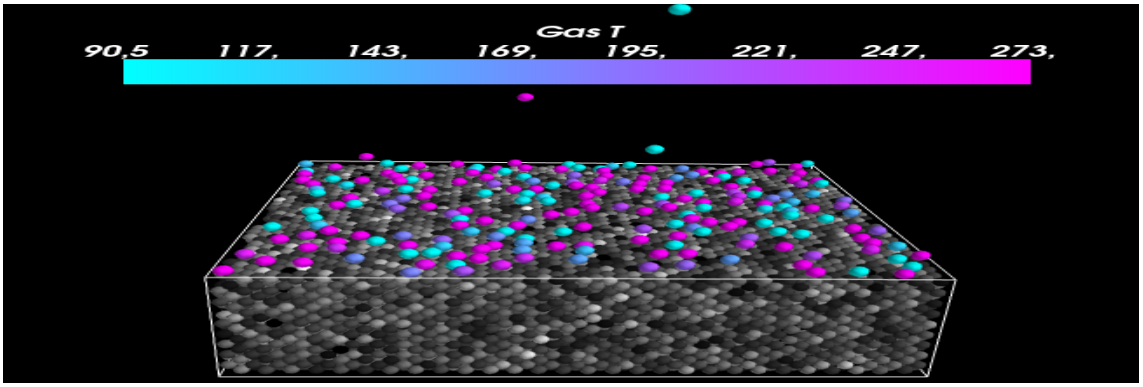

Рис. 12. Результат визуализации частиц и их температур, разделенных по типу, в заданной области

4.3. Визуализация

Для визуализации в данной работе использовались библиотеки Mayavi2 и Matplotlib. Для удобства реализации часто используемых способов, был написан модуль vis, который предоставляет обертку над методами из этих библиотек, комбинирующий их возможности для достижения заданного результата. Процесс визуализации данных на данный момент поддерживается только в однопоточном режиме в рамках одного процесса, однако mmdlab позволяет запустить такие задачи на множестве узлов или в многопроцессовом режиме. Например, рассмотрим задачу составления анимации, состоящей из кадров представляющих собой состояния исследуемой системы в последовательные моменты времени. Исходные данные распределены по множеству узлов с установленной серверной частью ParallelPython, то есть 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 визуализации; 2) собрать все отрисованные кадры на

мастер-узле; 3) собрать из этих кадров gif анимацию. На Рис. 13 приведен код, позволяющий выполнить такую задачу с использованием модуля mmdlab. Для сборки анимационного файла используется программа convert из пакета ImageMagick [12]. Так же возможно собирать результаты в видео-файл с использованием любого другого приложения (например, ffmpeg).

```
import sys
import mmdlab
nodes = ("10.0.0.1","10.0.0.2","10.0.0.3","10.0.0.4")
def read_and_vis(dir_state):
    from mmdlab.datareader.shortcuts import read_distr_gimm_data
    from mmdlab.vis import Points3d
    import mmdlab
    reader = read_distr_gimm_data(dir_state[0],
                                  "{}_*.dat".format(dir_state[1]))
    vis = Points3d(size=1, oncall=True, offscreen = True)
    frame = mmdlab.run([reader, vis])
    return (frame,dir_state[1])
def run_process(dir):
    from mmdlab.parallel import MultiPool
    from mmdlab.datareader.shortcuts import list_distr_gimm_states
    states = list_distr_gimm_states(dir)
    pool = MultiPool(1)
    results = pool.map(read_and_vis, [(dir,state) for state in states])
    return results
from mmdlab.parallel import once_per_node
results = once_per_node(nodes, run_process,
                        imports = "mmdlab", depfuncs=(read_and_vis,),
                        params = [(sys.argv[i+1],)
                                  for i,node in enumerate(nodes)])
for result in results:
   for frame, state in result:
        mmdlab.vis.savepng("./{}.png".format(state),frame)
mmdlab.vis.convert_to_gif("./*.png", delay_ms = 10)
```

Рис. 13. Сценарий для создания gif-анимации эволюции исследуемой системы во времени

5. IPython как средство управления и предварительного анализа

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

6. Заключение

В данной работе была сделана попытка создать высокоуровневую библиотеку, позволяющую легко кластеризировать и распараллеливать задачи чтения, обработки и визуализации. Основной задачей для данной библиотеки были анализ и визуализация данных, полученных в результате выполнения работы [1], и ее использование позволило в деталях рассмотреть эффект адсорбции азота на никелевой пластине (Рис. 14). Однако, особенности языка Python и архитектура этой библиотеки дают возможности легко расширить ее применение для обработки практически любых данных.

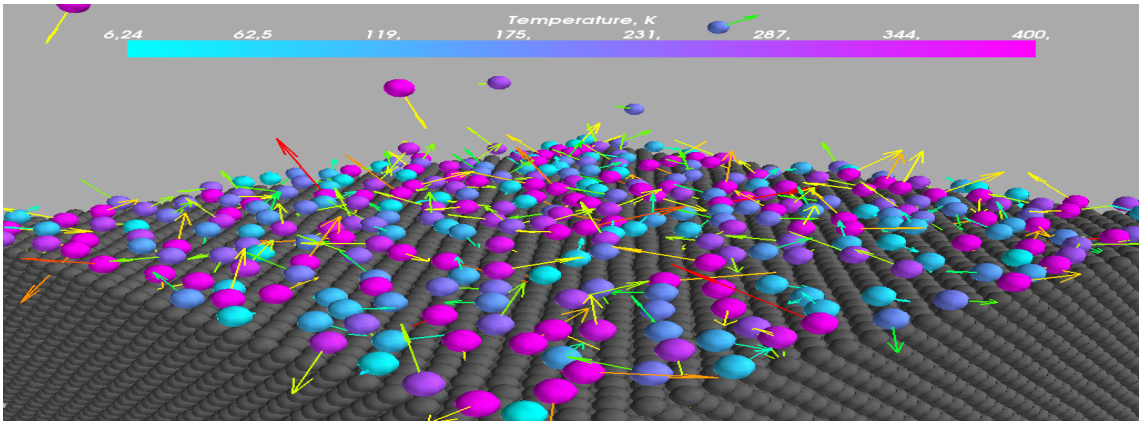

Рис. 14. Результат визуализации эффекта адсорбции на выбранном участке исследуемой системы

Литература

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.
2. Официальная документация Python. URL: https://www.python.org/ (дата обращения: 22.12.2015)
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. URL: http://ipython.org (дата обращения: 04.02.2016).
4. Официальная документация Numpy. URL: http://www.numpy.org/ (дата обращения: 04.02.2016).
5. Официальная документация Numba. URL: http://numba.pydata.org/ (дата обращения: 04.02.2016).
6. Официальная документация Cython. URL: http://cython.org/ (дата обращения: 04.02.2016).
7. Официальная документация Python, модуль Multiprocessing. URL: https://docs.python.org/2/library/multiprocessing.html (дата обращения: 04.02.2016).
8. Официальная документация ParallelPython. URL: http://www.parallelpython.com/ (дата обращения: 04.02.2016).
9. Официальная документация Mayavi2. URL: http://docs.enthought.com/mayavi/mayavi/mlab.html (дата обращения: 04.02.2016).
10. Официальная документация Matplotlib. URL: http://matplotlib.org/ (дата обращения: 04.02.2016).
11. Официальная документация Paramiko. URL: http://www.paramiko.org/ (дата обращения: 04.02.2016).
12. Официальная документация ImageMagick. URL: http://www.imagemagick.org/ (дата обращения: 04.02.2016).

Distributed visualization in application to the molecular dynamics simulation of equilibrium state in the gas-metal microsystems. *

D. Puzyrkov[1], V. Podryga[1], S. Polyakov[1]

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

In this paper a software package for processing and visualization of the molecular dynamics simulation is presented. During the development 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.

Keywords: visualization, molecular dynamics, cloud computing, parallel algorithms, Python, Mayavi2, Numba

References

1. Podryga V.O., Polyakov S.V., Puzyrkov D.V. Superkompiuternoe molekuliarnoe modelirovanie termodinamicheskogo ravnovesiia v mikrosistemakx gaz-metall [Supercomputer Molecular Modeling of Thermodynamic Equilibrium in Gas–Metal Microsystems] // Vychislitel'nye Metody i Programmirovanie [Numerical Methods and Programming]. 2015. T 16, No. 1. P. 123–138.
2. Python official documentation. URL: https://www.python.org/ (Date accessed: 22.12.2015).
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. URL: http://ipython.org (Date accessed: 04.02.2016).
4. Numpy official documentation. URL: http://www.numpy.org/ (Date accessed: 04.02.2016).
5. Numba official documentation. URL: http://numba.pydata.org/ (Date accessed: 04.02.2016).
6. Cython official documentation. URL: http://cython.org/ (Date accessed: 04.02.2016).
7. Python official documentation, Multiprocessing library. URL: https://docs.python.org/2/library/multiprocessing.html (Date accessed: 04.02.2016).
8. ParallelPython official documentation. URL: http://www.parallelpython.com/ (Date accessed: 04.02.2016).
9. Mayavi2 official documentation. URL: http://docs.enthought.com/mayavi/mayavi/mlab.html (Date accessed: 04.02.2016).
10. Matplotlib official documentation. URL: http://matplotlib.org/ (Date accessed: 04.02.2016).

*This work was supported by the Russian Foundation for Basic Researches (projects № 15-07-06082-a, № 15-29-07090-ofi_m)

11. Paramiko official documentation. URL: http://www.paramiko.org/ (Date accessed: 04.02.2016).

12. ImageMagick official documentation. URL: http://www.imagemagick.org/ (Date accessed: 04.02.2016).