МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра параллельных вычислений

Направление подготовки 09.03.01 Информатика и вычислительная техника

Направленность (профиль) Программная инженерия и компьютерные науки

ОТЧЕТ

о прохождении учебной практики, эксплуатационной практики
(указывается наименование практики)

Обучающегося Синюкова Валерий Константиновича группы №20203 курса 3
(Ф.И.О. полностью)

Тема задания: Формирование списка информации, необходимого для автоматической генерации LuNA-программ из последовательных

Место прохождения практики: Федеральное государственное бюджетное учреждение науки Институт вычислительной математики и математической геофизики Сибирского отделения Российской академии наук. Лаборатория Синтеза параллельных программ

Сроки прохождения практики: с 06.02.2023 г. по 27.05.2023 г.

Руководитель практики
от профильной организации Перепёлкин Владислав Александрович, н.с. ИВМиМГ СО РАН ________________
(Ф.И.О. полностью, должность) (подпись)

Руководитель практики от НГУ Малышкин Виктор Эммануилович, зав. каф. ПВ ФИТ, д.т.н., профессор ________________
(Ф.И.О. полностью, должность) (подпись)

Руководитель ВКР ____________________________________ ________________
(Ф.И.О. полностью) (должность)

Оценка по итогам защиты отчета: ____
(неудовлетворительно, удовлетворительно, хорошо, отлично)

Отчет заслушан на заседании кафедры ____
(наименование кафедры)

протокол __________**от** «______» _______________ 20_____г.

Новосибирск 2023

СОДЕРЖАНИЕ

Оглавление

ВВЕДЕНИЕ 4

ОПИСАНИЕ РАБОТЫ 5

1. Ознакомление с системой Sapfor 5
2. Схема автоматической генерации LuNA-программы 5
3. Формат списка знаний 5
4. Описание связи моделей вычислений традиционной параллельной программы и LuNA-программы 5
5. Пример списка знаний для конкретной последовательной программы 5

ЗАКЛЮЧЕНИЕ 6

Приложение А 7

6. Текстовое описание формата списка знаний 7

Список терминов 7

Список scalarDF 7

Список vectorDF 8

cmdParams 8

codeFragments 8

LuNAOperators 12

macrosCpp 14

macrosFa 14

Приложение Б 15

7. Описание связи моделей вычислений традиционной параллельной программы и LuNA-программы 15

LuNA-программа 15

Разбиение последовательной программы на фрагменты кода 15

Выделение фрагментов данных 19

Информационная зависимость и фрагменты данных единственного присваивания 21

Приложения В 23

8. Листинг программы, реализующей размытие в одномерном массиве на языке C++ ... 23

Приложение Г 24

9. .fa-файл программы, реализующей размытие в одномерном массиве на языке LuNA. 24
10. .cpp-файл программы, реализующей размытие в одномерном массиве на языке LuNA 25

Приложение Д 27

11. Список знаний, необходимый для автоматической генерации LuNA-программы, реализующей размытие в одномерном массиве 27

ВВЕДЕНИЕ

Цель практической работы: формирование списка информации, необходимого для автоматической генерации LuNA-программ, реализующих численные итерационные методы, с использованием системы Sapfor.

Задачи практической работы:

1. Установить, запустить, и проверить работу Sapfor+DVM на примере программы, реализующей итерационный метод Якоби решения систем дифференциальных уравнений (далее – программа Jacobi)
2. Проанализировать листинг параллельной программы Jacobi с использованием OpenMP и MPI, генерируемый системой DVMH.
3. Сформулировать перечень информации (список знаний), необходимой для автоматической генерации LuNA-версии программы Jacobi на основе написанной в рамках практики предыдущего семестра LuNA-программы
4. Описать связь моделей вычислений традиционной параллельной программы и LuNA-программы
5. Вручную написать пример LuNA-программы, реализующей какой-нибудь несложный алгоритм, а также пример списка знаний для этой программы

Актуальность темы задания: в данных момент существует множество систем автоматизации конструирования параллельных программ, среди них: MapReduce, Hadoop, Ahthill, Titanium, Unified Parallel и т.д. У каждой из данных систем есть своя область практического применения. Необходимо и дальше развивать методы автоматического конструирования параллельных программ с целью получения системы с наиболее широкой областью применения.

Место прохождения практики: Федеральное государственное бюджетное учреждение науки Институт вычислительной математики и математической геофизики Сибирского отделения Российской академии наук, Лаборатория синтеза параллельных программ.

Основные функции подразделения: изучение и развитие технологий параллельного программирования, перспективных систем параллельного программирования, теории и методов параллельного фрагментированного программирования.

Предполагаемые результаты практики: получение готового формата списка знаний, необходимых для автоматической генерации LuNA-программ, реализующих численные итерационные методы, описание связи моделей вычислений традиционной параллельной программы и LuNA-программы.

ОПИСАНИЕ РАБОТЫ

1. Ознакомление с системой Sapfor

   В рамках первого этапа работы была установлена система Sapfor, проведено ознакомление с ее функциональностью. После чего был проведен анализ листинга, генерируемого системой DVMH для программы Jacobi, в ходе которого был сделан вывод о том, что эта система генерирует параллельных код, используя объекты внутреннего представления, что значительно затрудняет анализ генерируемого листинга, поэтому этот этап работы был прекращен.

2. Схема автоматической генерации LuNA-программы

   Далее в первом приближении была составлена схема автоматической генерации LuNA-программ, которая выглядит следующим образом:

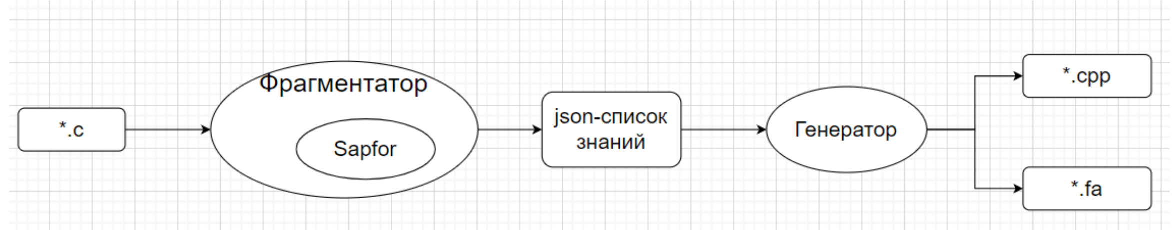


   Рисунок 1 – схема автоматической генерации LuNA-программы

   То есть выделяется некоторая сущность – *фрагментатор*, которая на вход должна получить последовательную программу на языке C, а выдать список знаний в формате json, который содержит всю необходимую информацию для автоматической генерации LuNA-программы, Sapfor является частью этого фрагментатора. Вышеописанный список знаний поступает на вход *генератору*, задача которого сгенерировать .cpp-файл с определениями фрагментов кода и .fa-файл с описателями фрагментов данных и фрагментов вычислений, то есть собственно говоря сгенерировать LuNA-программу.

3. Формат списка знаний

   Далее был начат основной этап практической работы – определение формата списка знаний. Формат был полностью определен, с ним можно ознакомиться в приложении А.

4. Описание связи моделей вычислений традиционной параллельной программы и LuNA-программы

   Далее, в следствие того, что формулировка списка знаний опирается на модель вычислений, используемую в системе LuNA, было составлено описание связи моделей вычислений традиционной параллельной программы и LuNA-программы. С данным описанием можно ознакомиться в приложении Б.

5. Пример списка знаний для конкретной последовательной программы

   После того, как первая версия json-формата была зафиксирована, были написаны программы на языке C++ и на языке LuNA, реализующие размытие в одномерном массиве, а также json-список знаний (составленный по вышеописанному формату) для данной последовательной программы. Основная цель данной работы заключается в создании собственно примера json-объекта, составленного по вышеописанному формату, который должен способствовать лучшему восприятию самого формата. Кроме того, мы в первом приближении смогли проверить пригодность формата для дальнейшего использования и исправить небольшие недочеты. С версией данной программы на языке C++ можно ознакомиться в приложении В, на языке LuNA – в приложении Г, со списком знаний – в приложении Д.

ЗАКЛЮЧЕНИЕ

В результате данной работы было проведено знакомство с системой Sapfor, а также с кодом, который генерирует система DVMH. В первом приближении была составлена схема автоматической генерации LuNA-программы. Основной частью работы стало определение формата списка информации, необходимой для автоматической генерации LuNA-программы, этот формат был полностью определен. Было составлено описание связи моделей вычислений традиционной параллельной программы и LuNA-программы. Были написаны программы на языках C++ и LuNA, реализующие размытие в одномерном массиве, для этих программ был написан пример списка знаний для более наглядного представления вышеописанного формата. Таким образом, все задачи, которые поставлены в рамках данной практической работы, были выполнены.

Приложение А

6. Текстовое описание формата списка знаний

Список терминов

- `length(arr)` - количество элементов в массиве arr.
- `sum(arr)` - сумма всех элементов массива arr.

Объект верхнего уровня — это словарь, содержащий 7 списков:

```
{
	"scalarDF": [...],
	"vectorDF": [...],
	"cmdParams": [...],
	"CF": [...],
	"LuNAOperators": [...],
	"MacrosCpp": [...],
	"MacrosFa": [...],
}
```

- `scalarDF` — список описателей скалярных фрагментов данных
- `vectorDF` — список описателей векторных фрагментов данных
- `cmdParams` — список параметров “командной строки”
- `CF` — список описателей фрагментов кода,
- `LuNAOperators` — список описателей операторов LuNA-программы
- `MacrosCpp` — список макросов C++
- `MacrosFa` — список макросов на языке LuNA

Список scalarDF

В списке перечислены описатели скалярных фрагментов данных (под скалярными следует понимать фрагменты данных, которые в последовательной программе были переменными-”не массивами”безындексные). Каждый описатель — это объект, содержащий 2 строковых поля:

```
{
	"scalarDF": [ {
		"name": "<имя>",
		"type": "int" | "double" | "string"
	}...
	...
}
```

- **`name`** - строковое название описателя фрагмента данных, на него накладываются те же ограничения, что и на имена переменных в традиционных языках программирования.
- **`type`** - тип ФД, может иметь одно из следующих значений "int", "double" или "string".

Список vectorDF

Список содержит описатели векторных фрагментов данных (под векторными следует понимать фрагменты данных, которые в последовательной программе были переменными-”массивами”имеющих индексы) и устроен аналогично scalarDataFragments, кроме того, что у элементов списка есть еще поле axes - список, хранящий массив целых чисел, размер которого равен размерности (количеству индексов) фрагмента данных. Значение axes[i] - размер i-ой оси фрагмента данных. Элементом списка axes может быть либо целое число, либо строковый литерал, соответствующий имени переменной/макроса, значение которой(ого) равно размеру данной оси массива.

```
{
      "vectorDF": [ {
            "name": "<имя>",
            "type": "int" | "double" | "string",
            "axes": [<int> | "<name>", ...]
      }...
      ...
}
```

cmdParams

Список устроен аналогично scalarDataFragments и соответствует параметрам передаваемым в фрагментированную подпрограмму (см. https://gitlab.ssd.sscc.ru/luna/luna/-/wikis/luna_lang_v01) main.

codeFragments

Элементы данного списка соответствуют фрагментам кода и содержат следующие элементы:

- **name** - имя с++-функции, соответствующей данному фрагменту кода.
- **start, end** - списки одинаковой длины, элементами которых являются соответственно первые и последние символы участков кода в файле с исходной программе, которые должны быть включены в данный фрагмент кода.
- **inTypes/outTypes** - упорядоченный массив строковых литералов, каждый элемент которого соответствует типу входного/выходного параметра функции, соответствующей данному фрагменту кода и может принимать одно из следующих значений: “int”, ”double”, ”string”.
- **inTypesV/outTypesV** - упорядоченный массив boolean-флагов, выставленный флаг inTypesV[i]/outTypesV[i] сигнализирует о том, что параметр, соответствующий inTypes[i]/outTypes[i], является векторным, в противном случае он является скалярным.
- **in/out** - упорядоченный массив уникальных строковых литералов, in[i]/out[i] соответствует имени параметра с типом inTypes[i]/outTypes[i] функции, соответствующей данному фрагменту кода. Если элемент присутствует как в in, так и out, то он считается одновременно входным и выходным. На значения элементов in/out накладываются те же

ограничения, что и на имена переменных в традиционных языках программирования.
!!! length(inTypes) = length(inTypesV) = length(in) и length(outTypes) = length(outTypesV) = length(out)

- **`fragmented`** - список целочисленных значений, размер которого равен размерности фрагментации данного фрагмента кода. В случае, если этот список пустой, то фрагментация не происходит. Элементами данного списка являются номера осей, по которой происходит фрагментация. Для генератора это, во-первых, значит то, что в сигнатуру функции, соответствующей данному фрагменту кода, должны быть добавлены номер текущего фрагмента и количество фрагментов всего для каждой оси. Данные переменные будут иметь тип int и называться f$i и nf$i соответственно, где $i - номер оси. Например, если фрагментация происходит с сеткой 2x2, то в сигнатуру функции будут добавлены f0, nf0, f1, nf1. Также генератор должен будет выполнить пересчет границ фрагментированных циклов и поменять индексацию массива. Чтобы это сделать, ему потребуется поле fragmentedLoopsNest, описанное далее (если размер массива fragmented равен нулю, то поле fragmentedLoopsNest игнорируется). Порядок элементов в данном списке значения не имеет, но должен быть зафиксирован, так как влияет на интерпретацию дальнейших полей.
- **`fragmentedLoopsNest`** - упорядоченный список json-словарей, каждый элемент которого соответствует одному гнезду циклов внутри фрагмента кода и содержит следующие поля:
    - **`inductives`** - упорядоченный список строковых литералов, i-ый элемент которого соответствует имени индуктивной переменной i-го цикла внутри гнезда (на данном этапе считается, что порядок циклов внутри гнезда и порядок осей массива совпадают, то есть
    ```
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            for (int k = 0; k < l; ++k)
                calc(array[i][j][k]);
    ```
    является правильной последовательностью, а
    ```
    for (int k = 0; k < l; ++k)
        for (int j = 0; j < m; ++j)
            for (int i = 0; i < n; ++i)
                calc(array[i][j][k]);
    ```
    не является правильной последовательностью, это нужно для того, чтобы).
    - **`inits`/`ends`** - упорядоченный список строковых литералов или целочисленных значений (в одном списке может быть и то, и другое), i-ый элемент которого соответствует начальному/(конечному) значению индуктивной переменной i-го цикла. Если элемент списка это строковый литерал, то он также

должен принадлежать списку in данного фрагмента кода и иметь тип int, иначе описание считается составленным неверно.
Для вышеописанного примера:

```
inductives = [“i”,”j”,”k”]
inits = [0,0,0]
ends = [“n”,”m”,”l”]
```

- **start** - номер первого символа гнезда цикла в исходном файле.
- **end** - номер последнего символа гнезда цикла в исходном файле.
- **body_start** - номер первого символа тела гнезда цикла в исходном файле.
- **body_end** - номер последнего символа тела гнезда цикла в исходном файле.

С этими данными (fragmented и вышеописанный полями fragmentedLoopsNest) генератор будет выполнять следующие действия. Во-первых, в тело фрагмента кода будут вставлены вычисления начальных и конечных значений индуктивных переменных для каждой оси, по которой происходит фрагментация. Далее из фрагмента кода будут вырезаны все гнезда циклов, информация о которых есть в fragmentedLoopsNest. Вместо них будут вставлены новые гнезда циклов, составленные следующим образом. Количество, порядок и индуктивные переменные циклов в гнезде будут такие же, как в оригинальном гнезде. Для всех циклов, по которым происходит фрагментация, верхняя граница индуктивной переменной будет заменена на вычисленное ранее конечное значение. Тело цикла останется без изменений, за исключением изменения оператора if (в случае его присутствия в теле гнезда), в котором ко всем индуктивным переменным должны быть прибавлены начальные значения этой переменной (для этого они и вычисляются).
Пример:

- Исходная программа:

```
void calc_A(float ***A, int n, int m,int l)
{
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            for (int k = 0; k < l; ++k)
                if (i != 0 && j != 0 && k != 0)
                    calc(A[i][j][k]);
}
```

- Фрагмент кода:

```
int get_start(int size, int f, int nf)
{
    return f * size / nf;
}

int get_size(int size, int f, int nf)
{
    return (f + 1) * size / nf - f * size / nf;
}

void calc_A_fragmented(int f0, int nf0, int f1, int nf1, int f2, int nf2, InputDF&A, int n, int m, int l)
{
    int start0 = get_start(n, f0, nf0);
    int size0 = get_size(n, f0, nf0);
    int start1 = get_start(n, f1, nf1);
    int size1 = get_size(n, f1, nf1);
    int start2 = get_start(n, f2, nf2);
    int size2 = get_size(n, f2, nf2);

    for (int i = 0; i < size0; ++i)
        for (int j = 0; j < size1; ++j)
            for (int k = 0; k < size2; ++k)
                if (i + start0 != 0 && j = start1 != 0 && k + start2 != 0)
                    calc(A[i][j][k]);
}
```

Далее продолжается описание полей fragmentedLoopsNest, связанное с теневыми гранями.

- **shadow** - json-словарь, который содержит информацию о том, какие теневые грани потребуются данному фрагменту кода для вычислений тела данного гнезда циклов, либо какие теневые грани данный фрагмент кода должен инициализировать, либо и то, и другое.
    - Ключом является строковый литерал, соответствующий имени входного векторного фрагмента данных данного фрагмента кода.
    - Значением данного словаря является неупорядоченный список, каждый элемент которого соответствует одному “потенциальному доступу к теневым граням“ внутри данного гнезда циклов (см. “потенциальный доступ к теневым граням” на рисунке) и является упорядоченным списком, i-ый элемент которого является расстоянием зависимостей по i-ой оси данного массива у данного потенциального доступа к теневым граням.

Пример (потенциальный обращения к теневым граням обозначены красным цветом):

```
void countAB(float **A, float *B, int n, int m)
{
    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < m; ++j)
        {
            if (2 < i && i < m - 2 && 2 < j && j < m - 3 && 10 != A[i - 2][j + 3])
            {
                A[i][j] = A[i - 1][j] + A[i - 2][j + 2] + A[i + 2][j + 2];
                B[i] += A[i][j];
            }
        }
        B[i] *= 10;
    }
}
```

Для этого гнезда циклов значение shadow будет следующим:

```
"shadow" :
{
  "A" : [[-2, 3], [-1, 0], [-2, 2], [2, 2]]
}
```

При условии, что значение fragmented следующее:

```
"fragmented": [0, 1]
```

Для генератора это в числе прочего означает то, что сигнатура данного фрагмента кода должна быть расширена, в нее должно быть добавлены параметры типа InputDF&, которые соответствуют принимаемым теневым граням, они именуются A_sh_i_j_…_from, где (i,j,...) - расстояние от отправляющего фрагмента вычислений до принимающего фрагмента вычислений (то есть если топология фрагментирования двумерная и теневая грань должна быть получена от фрагмента вычислений, находящегося на одну строчку выше, то i = 1, j = 0), в случае если расстояние отрицательное, то к соответствующей компоненты добавляется префикс m. Должно быть получено столько же параметров типа OutputDF&, которые соответствуют инициализируемым этим фрагментом кода теневым граням, они именуются так же, как и параметры, соответствующие входным теневым граням, только суффикс “from” меняется на суффикс “to”.

LuNAOperators

неупорядоченный список LuNA-операторов

- `type` - может принимать одно из следующих значений: “computantinalFragment” или “loop”. Данное поле определяет практически все поля данного элемента списка LuNA-операторы.
  Далее указаны, каким типам операторов какие поля присущи.

computationalFragment

- **`CF`** - имя фрагмента кода, который будет вызываться.
- **`args`** - упорядоченный массив строковых литералов, соответствующих аргументам фрагмента вычислений. i-ый элемент данного массива соответствует i-ому входному параметру фрагмента кода $CF, если i < length(in), и (i - length(in)) выходному параметру фрагмента кода $CF иначе, и должен принадлежать списку in или out соответственно. Зафиксируем i < length(in). i-ый элемент данного массива должен являться ключом в scalarDF, если inTypesV[i] = false для фрагмента кода $CF, и vectorDF, если inTypesV[i] = true. Тип фрагмента данных, указанный в соответствующей записи scalarDF или vectorDF, должен совпадать с типом inTypes[i]. Для i >= length(in) условия такие же с точностью до замены в предыдущих замечаниях “in” на “out”.
- **`fragmented`** - упорядоченный список, i-ый элемент которого указывает на то, насколько фрагментов по i-ой оси должно происходить разбиение данного фрагмента вычислений, и может быть либо целым положительным числом, либо строковым литералом, равным имени фрагмента данных либо макроса, “содержащего” число фрагментов. Для генератора это означает то, что данный фрагмент вычислений необходимо обрамить гнездом LuNA-операторов for, в котором будет столько операторов for, сколько существует отличных от единицы элементов списка fragmented (данные LuNA-операторы соответствуют фрагментации по пространству). Допустим, что fragmented[i] = k, причем fragmented[i] является j-ым не равным единице элементом списка fragmented. тогда j-ый цикл гнезда будет выглядеть следующим образом:

```
for $CF_inductive_j=0..k-1
    …
```

- **`ordered`** - boolean-флаг, который указывает на то, что фрагментированные вычисления должны выполняться строго в порядке возрастания индексов LuNA-операторов for, соответствующих фрагментации по пространству. Необходимо, например, когда в фрагментированных фрагментах кода происходит вывод, который должен быть строго упорядочен. В случае, если все элементы fragmented равны 1, то данное поле игнорируется.
- **`reduction`** - неупорядоченный список, у элементов которого есть следующие поля:
    - **`name`** - Имя переменной, должно присутствовать в outputFD фрагмента кода $CF.
    - **`operation`** - Название редукционной операции - строковый литерал, который принимает одно из следующих значений: *max*, *min*, *avg*, *sum*, *prod* (и т.д.).

  Каждый элемент данного списка соответствует одной редукционной операции, которую нужно выполнить после выполнения всех “фрагментированных” (см. поле fragmented) фрагментов вычислений.

Если фрагментация данного фрагмента вычислений не происходит, то есть все элементы списка fragmented равны 1, то поле reduction будет проигнорировано генератором.

loop

Соответствует фрагментации по времени.

- **`inductive`** - строковый литерал, имя индуктивной переменной, на которое накладываются те же ограничения, что и на имена переменных в традиционных языках программирования.
- **`start/end`** - значение индуктивной переменной, может быть либо целым числом, либо строковым литералом, соответствующему имени фрагмента вычислений либо макроса, “содержащего” целочисленное значение, на котором цикл должен начаться/”закончиться”.
- **`operators`** - json-список, который устроен полностью аналогично устройству LuNAOperators и соответствует LuNA-операторам, находящимся в теле данного оператора for.
- **`breaks`** - список, элементы которого содержат следующие поля:
  - **`start`** - номер символа в исходном файле, с которого начинается условие break данного цикла
  - **`end`** - номер символа в исходном файле, на котором заканчивается условие break данного цикла

macrosCpp

Неупорядоченный список, элементы которого содержат следующие поля:

- **`start/end`** - номер первого/последнего символа данного макроса в файле исходной программы.

macrosFa

Устроены аналогично macrosCpp, только соответствующие макросы будут вставлены в .fa-файл.

Приложение Б

7. Описание связи моделей вычислений традиционной параллельной программы и LuNA-программы

LuNA-программа

LuNA-программа - это множество информационно зависимых задач и множество перемежающихся между этими задачами данных. Эти информационно зависимые задачи в LuNA называются **фрагментами вычислений**, а данные - **фрагментами данных**. У каждого фрагмента вычислений есть реализация в виде c++-функции, эта реализация называется **фрагментом кода**. Фрагмент кода соотносится с фрагментом вычислений так же, как процедура соотносится с вызовом процедуры. Фрагменты вычислений и фрагменты данных описываются в **.fa**-файле (можно сказать, что это и есть LuNA-программа). Фрагменты кода описываются в отдельном .cpp-файле.

Разбиение последовательной программы на фрагменты кода

Для того, чтобы разбить последовательный код на фрагменты кода, необходимо понимать, что фрагмент вычислений - это некоторый код, который будет выполняться на одном вычислительном узле, то есть последовательно. Соответственно, тот код, которые мы хотим, чтобы выполнялся параллельно, должен выполняться в разных фрагментах вычислений. Для лучшего понимания рассмотрим следующий пример:

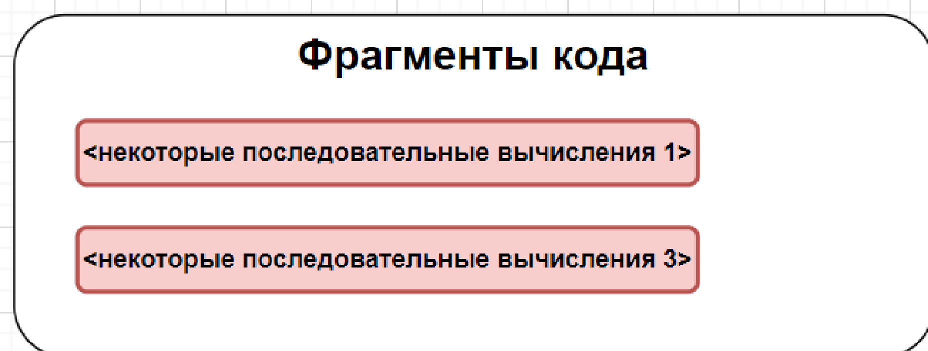
Пусть необходимо создать LuNA-программу из следующей последовательной программы:

```
...
int main(int argc, char* argv[])
{
	<некоторые последовательные вычисления 1>

	for (int i = 0; i < N; ++i)
		for (int j = 0; j < M; ++j)
			<некоторые последовательные вычисления 2>

	<некоторые последовательные вычисления 3>

	return 0;
}
```

Последовательно разберем, какие фрагменты кода необходимо выделить.

1. Вряд ли можно получить какую-то выгоду от распараллеливания последовательных вычислений 1 или последовательных вычислений 3, так что каждые из этих вычислений могут выполняться в рамках одного и того же фрагмента вычислений. То есть мы можем выделить два фрагмента кода, первый содержит код “некоторых последовательных вычислений 1”, второй - “некоторых последовательных вычислений 3”.

```
...
int main(int argc, char* argv[])
{
<некоторые последовательные вычисления 1>

for (int i = 0; i < N; ++i)
 for (int j = 0; j < M; ++j)
  <некоторые последовательные вычисления 2>

<некоторые последовательные вычисления 3>

return 0;
}
```

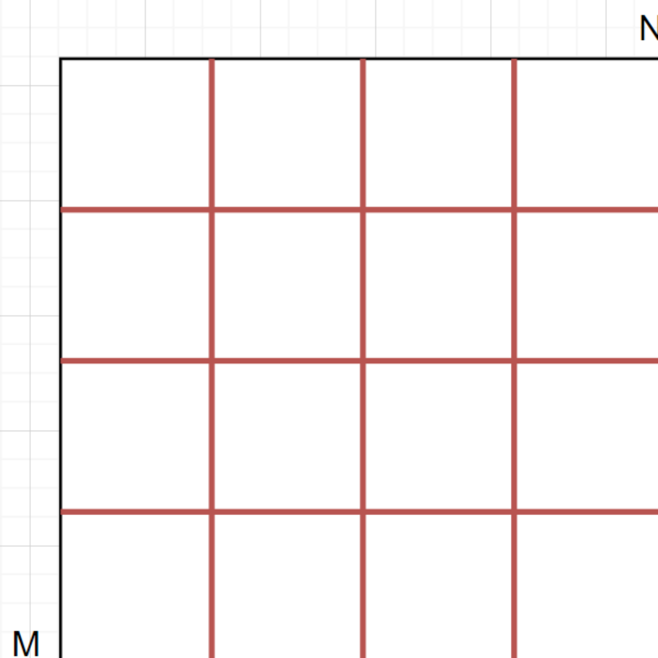


2. Теперь самое интересное: как нам разбить гнездо циклов на фрагменты кода. Что будет, если просто поместить весь цикл в один фрагмент кода?

```
...
int main(int argc, char* argv[])
{
 <некоторые последовательные вычисления 1>

 for (int i = 0; i < N; ++i)
  for (int j = 0; j < M; ++j)
   <некоторые последовательные вычисления 2>

 <некоторые последовательные вычисления 3>

 return 0;
}
```

Так можно сделать, и программа будет работать корректно, только очевидно, что выигрыша в производительности получено не будет, потому что большая часть вычислительной сложности данной программы заключена в этом гнезде циклов, и мы хотим его вычислять параллельно, а если выделить фрагмент кода вышеописанным способом, то данное гнездо циклов будет вычисляться последовательно.

3. Как быть? Нужно фрагментировать вычисления, то есть разбить данный цикл на несколько блоков и разные блоки выполнять в разных фрагментах вычислений. Для определенности возьмем топологию фрагментирования 4x4, то есть разбивать будем всего на 16 блоков.



4. Как перенести это в код?

a. В .cpp-файле создается следующий фрагмент кода:

```
void code_fragment_loop(int f0, int nf0, int f1, int
nf1, int N, int M, …)
{
	int start0 = get_start(f0,nf0,N);
	int start1 = get_start(f1,nf1,M);
	int size0 = get_size(f0,nf0,N);
	int size1 = get_size(f1,nf1,M);

	for (int i = start0; i < size0; ++i)
		for (int j = start1; j < size1; ++j)
			<некоторые последовательные вычисления
		2>
}
```

где *f0, f1, nf0, nf1* - номер фрагмента вычислений по 0 оси, номер фрагмента вычислений по 1 оси, количество фрагментов вычислений по 0 оси и количество фрагментов вычислений по 1 оси соответственно, функции get_size и get_start вычисляют границы циклов для данного фрагмента вычислений.

b. В .fa-файле данный фрагмент кода вызывается следующим образом:

```
...
for i=0..3
	for j=0..3
		code_fragment_loop(i,4,j,4,N,M,...);
```

Подобное фрагментирование будем далее называть **фрагментированием по пространству**.

5. В итоге получается 3 фрагмента кода:

а LuNA-программа, то есть .fa-файл, выглядит следующим образом (фрагменты кода, содержащие некоторые последовательные вычисления 1 и некоторые последовательные вычисления 3 назовем code_fragment1 и code_fragment3 соответственно):

```
…
code_fragment1(...);


for i=0..3
     for j=0..3
          code_fragment_loop(i,4,j,4,N,M,...);


code_fragment3(...);
```

Выделение фрагментов данных

Чуть усложним наш пример из предыдущего параграфа, добавим переменную x, значение которой будет инициализировано в “некоторых последовательных вычислениях 1”, а использоваться будет в некоторых последовательных вычислениях 2 и 3.

```
...
int main(int argc, char* argv[])
{
int x;
 <некоторые последовательные вычисления 1, инициализация x>

 for (int i = 0; i < N; ++i)
  for (int j = 0; j < M; ++j)
   <некоторые последовательные вычисления 2 с переменной x>

 <некоторые последовательные вычисления 3 с переменной x>

 return 0;
}
```

Очевидно, что просто создавать заново переменную x в каждом из трех выделенных фрагментов кода нельзя, так как логика программы будет нарушена. Возникает идея: нужно выделить x на более верхнем уровне и каким-то образом передавать ее в фрагменты вычислений, соответствующие последовательным вычислениям. В LuNA так и делается, более высокий уровень - это .fa-файл, в котором выделяется *фрагмент данных* x. Фрагмент данных может передаваться в фрагмент вычислений, как входной, то есть необходимый для вычисления фрагмента вычислений и как выходной, то есть инициализируемый в фрагменте вычислений. В нашем случае получается, что x будет передаваться в code_fragment1 в качестве выходного фрагмента данных, а в code_fragment3 и code_fragment_loop в качестве входного фрагмента данных.

Получается, что фрагменты кода в нашей программе будут следующие:

```
void code_fragment1(OutputDF& x, …)
{
    <последовательные вычисления 1, инициализация x>
}
```

(тип параметра OutputDF& в сигнатуре фрагмента кода соответствует выходному фрагменту данных)

```
void code_fragment3(int x, …)
{
    <последовательные вычисления 3 с переменной x>
}
```

(тип параметра, соответствующего входному фрагменту данных, - InputDF&. В случае, если фрагмент данных скалярный, можно указать конкретный тип, как сделано в сигнатуре code_fragment3)

```
void code_fragment_loop(int f0, int nf0, int f1, int nf1, int
N, int M, int x, …)
{
    int start0 = get_start(f0,nf0,N);
    int start1 = get_start(f1,nf1,M);
    int size0 = get_size(f0,nf0,N);
    int size1 = get_size(f1,nf1,M);

    for (int i = start0; i < size0; ++i)
        for (int j = start1; j < size1; ++j)
            <некоторые последовательные вычисления 2 с
        переменной x>
}
```

А LuNA-программа (то есть .fa-файл) выглядит следующим образом:

```
…
df x;
code_fragment1(x, ...);


for i=0..3
    for j=0..3
```

```
        code_fragment_loop(i,4,j,4,N,M,x, ...);

code_fragment3(x, ...);
```

Из данного параграфа можно сделать следующий вывод: программисту, который “переводит” последовательную программу на язык LuNA в числе прочих необходимо решить задачу о том, какие переменные могут остаться локальными переменными внутри фрагментов кода, а какие должны “превратиться” в фрагменты данных, выделяемые в .fa-программе, и передаваться в фрагменты вычислений. Решение это принимается относительно несложно при условии, что разбиение последовательной программы на фрагменты кода зафиксировано: если переменная используется только в одном фрагменте кода, то она остается локальной внутри этого фрагмента кода, а если переменная используется в двух и более фрагментах кода, то она должна “превратиться” в фрагмент данных. Но на этом все задачи, связанные с выделением фрагментов данных не заканчиваются.

Информационная зависимость и фрагменты данных единственного присваивания

В преамбуле было отмечено, что “LuNA-программа - это множество информационно-зависимых задач”, сейчас мы разберемся, что это значит. В LuNA фрагмент данных может находится в двух состояниях: вычисленный и не вычисленный. Фрагмент данных становится вычисленным после того, как завершается фрагмент вычислений, в который этот фрагмент данных передавался как выходной, до тех пор с момент своего создания он считается невычисленным. Зная это, информационная зависимость выражается очень просто: состояния, в которых может находится фрагмент вычислений, в некотором смысле похожи на состояния, в которых может находится процесс в операционной системе. Если все входные фрагменты данных фрагмента вычислений вычислены, то он считается готовым к вычислению и в какой-то момент в будущем будет вычислен. Если какие-то входные фрагменты данных фрагмента вычислений не вычислены, то он ожидает (спит) до тех пор, пока все его входные фрагменты данных станут вычисленными.

Отсюда вытекает еще один момент, понимание которого необходимо для написания LuNA-программ: фрагменты данных - **единственного присваивания**, то есть один и тот же фрагмент данных не может быть выходным для двух и более фрагментов вычислений. Это свойство фрагментов данных очевидно необходимо для выражения информационной зависимости.

Разберем, как это влияет на конструирование LuNA-программ на примере.

Пусть имеется следующий участок последовательного кода:

```
for (int i = 0; i < T; ++i)
```

```
calc_eps(&eps, ...);
```

И пусть по тем или иным соображениям, мы не хотим этот цикл переносить в отдельный фрагмента кода, а хотим “моделировать” его с помощью LuNA-оператора for:

```
df T, eps;
…
for i=0..T-1
     calc_eps(eps,...);
```

где cals_eps() - выделенный нами фрагмент кода, соответствующий calc_eps() исходной программы.

Так вот, так как сделано выше, делать нельзя, так как фрагменты данных - единственного присваивания!

Как быть? Идея: пусть eps будет не скалярным, а векторным фрагментом данных и на каждой итерации цикла мы будем передавать в calc_eps() следующий элемент массива eps:

```
df T, eps;
…
for i=0..T-1
     calc_eps(eps[i], eps[i-1], ...);
```

где сигнатура calc_eps() следующая:

```
calc_eps(OutputDF& current_eps, InputDF& prev_eps, …)
```

(В данном примере мы сделали предположение, что для вычисления следующего значения eps необходимо предыдущее, именно поэтому в сигнатуре функции присутствует параметр prev_eps, в противном случае можно было бы обойтись без него).

Подобный прием называется **фрагментацией по времени**.

Приложения В

8. Листинг программы, реализующей размытие в одномерном массиве на языке C++

```
#define N 100000
#define T 10000

#include <stdio.h>

int main(int argc, char *argv[])
{
    double *A = new double[N];
    for (int i = 0; i < N; ++i)
        A[i] = 2 * i + (i % 10);

    for (int t = 0; t < T; ++t)
        for (int i = 1; i < N - 1; ++i)
            A[i] = (A[i - 1] + A[i] + A[i + 1]) / 3;

    for (int i = 0; i < N; ++i)
        printf("%11.3f\n", A[i]);

    delete A;
    return 0;
}
```

Приложение Г

9. .fa-файл программы, реализующей размытие в одномерном массиве на языке LuNA

```
#define T 10000

import init_A(int, int, name, name, name) as init_A;
import blur(int, int, value, value, value, name, name, name) as blur;
import print_result(int, int, value) as print_result;
import init_extreme_border(name) as init_extreme_border;

C++ sub empty() ${{ $}}

// nf - number of fragments
sub main(int nf) {
    df A, A_shadow_1, A_shadow_m1, print;

    for f=0..nf-1
        init_A(nf, f, A[0][f], A_shadow_1[0][f + 1],A_shadow_m1[0][f + 1]);

    for t=0..$T-1
    {
        init_extreme_border(A_shadow_m1[t][0]);
        init_extreme_border(A_shadow_1[t][nf + 1]);
        for f=0..nf-1
            blur(nf, f, A[t][f], A_shadow_1[t][f + 2], A_shadow_m1[t][f],
                A[t + 1][f], A_shadow_1[t + 1][f + 1], A_shadow_m1[t + 1][f + 1]);
    }

    empty() >> (print[0]);

    for f=0..nf-1
        if (print[f])
            print_result(nf, f, A[$T][f]) >> (print[f + 1]);
}
```

10. .cpp-файл программы, реализующей размытие в одномерном массиве на языке LuNA

```
#define N 100000

#include <stdio.h>

#include "ucenv.h"

extern "C"
{

    int get_size0(int nf0, int f0)
    {
        return (f0 + 1) * N / nf0 - f0 * N / nf0;
    }

    int get_start0(int nf0, int f0)
    {
        return f0 * N / nf0;
    }

    void create_shadows(int size0, double * A_arr, OutputDF& A_shadow_1, OutputDF&
A_shadow_m1)
    {
        // left shadow border
        double *A_shadow_m1_arr = A_shadow_1.create<double>(1);
        // right shadow border
        double *A_shadow_1_arr = A_shadow_m1.create<double>(1);

        for (int i = 0; i < 1; ++i)
            A_shadow_m1_arr[i] = A_arr[i + 1];

        for (int i = 0; i < 1; ++i)
            A_shadow_1_arr[i] = A_arr[size0 + i];
    }

    void init_A(int nf0, int f0, OutputDF &A, OutputDF& A_shadow_1, OutputDF&
A_shadow_m1)
    {
        int start0 = get_start0(nf0, f0);
        int size0 = get_size0(nf0, f0);

        // + 2 - for shadow borders
        double *A_arr = A.create<double>(size0 + 2);

        // + 1 - because of shadow borders
        for (int i = start0 + 1; i < start0 + size0 + 1; ++i)
            A_arr[i - start0] = 2 * (i - 1) + ((i - 1) % 10);

        create_shadows(size0, A_arr, A_shadow_1, A_shadow_m1);
    }
```

```
    void init_extreme_border(OutputDF& extreme_border)
    {
        extreme_border.create<double>(1);
    }

    void blur(int nf0, int f0, InputDF &A_prev, InputDF &A_prev_shadow_1, InputDF
&A_prev_shadow_m1,
        OutputDF &A, OutputDF& A_shadow_1, OutputDF& A_shadow_m1)
    {
        int start0 = get_start0(nf0, f0);
        int size0 = get_size0(nf0, f0);

        A = A_prev;
        double *A_arr = (double *)(static_cast<const size_t *>(A.get_data()));
        double *A_prev_shadow_1_arr = (double *)(static_cast<const size_t
*>(A_prev_shadow_1.get_data()));
        double *A_prev_shadow_m1_arr = (double *)(static_cast<const size_t
*>(A_prev_shadow_m1.get_data()));

        // inserting shadow borders
        for (int i = 0; i < 1; ++i)
            A_arr[i] = A_prev_shadow_m1_arr[i];
        for (int i = 0; i < 1; ++i)
            A_arr[size0 + 1 + i] = A_prev_shadow_1_arr[i];

        int is_first = f0 == 0;
        int is_last = f0 == (nf0 - 1);
        for (int i = start0 + 1 + is_first; i < start0 + size0 + 1 - is_last; ++i)
            A_arr[i - start0] = (A_arr[i - start0 - 1] + A_arr[i - start0] +
A_arr[i - start0 + 1]) / 3;

        create_shadows(size0, A_arr, A_shadow_1, A_shadow_m1);
    }

    void print_result(int nf0, int f0, InputDF& A)
    {
        int start0 = get_start0(nf0, f0);
        int size0 = get_size0(nf0, f0);

        double *A_arr = (double *)(static_cast<const size_t *>(A.get_data()));

        // + 1 - because of shadow borders
        for (int i = start0 + 1; i < start0 + size0 + 1; ++i)
            printf("%11.3f\n", A_arr[i - start0]);
    }
}
```

Приложение Д

11. Список знаний, необходимый для автоматической генерации LuNA-программы, реализующей размытие в одномерном массиве


```
{
    "scalarDF": [],
    "vectorDF": [
        {
            "name": "A",
            "type": "double",
            "axes": [
                "N"
            ]
        }
    ],
    "cmdParams": [
        {
            "name": "nf",
            "type": "int"
        }
    ],
    "codeFragments": [
        {
            "name": "init_A",
            "start": [
                132
            ],
            "end": [
                192
            ],
            "inTypes": [],
            "inTypesV": [],
            "in": [],
            "outTypes": [
                "double"
            ],
            "outTypesV": [
                true
            ],
            "out": [
                "A"
            ],
            "fragmented": [
                0
            ],
            "fragmentedLoopsNest": [
                {
                    "inductives": [
                        "i"
                    ],
                    "inits": [
```

```
                0
            ],
            "ends": [
                "N"
            ],
            "start": 132,
            "end": 192,
            "body_start": 169,
            "body_end": 192
        }
    ],
    "shadow": {
        "A": [
            [
                -1
            ],
            [
                1
            ]
        ]
    }
},
{
    "name": "blur",
    "start": [
        238
    ],
    "end": [
        322
    ],
    "inTypes": [
        "double"
    ],
    "inTypesV": [
        true
    ],
    "in": [
        "A"
    ],
    "outTypes": [
        "double"
    ],
    "outTypesV": [
        true
    ],
    "out": [
        "A"
    ],
    "fragmented": [
        0
```

```
        ],
        "fragmentedLoopsNest": [
            {
                "inductives": [
                    "i"
                ],
                "inits": [
                    0
                ],
                "ends": [
                    "N"
                ],
                "start": 238,
                "end": 322,
                "body_start": 283,
                "body_end": 322
            }
        ],
        "shadow": {
            "A": [
                [
                    -1
                ],
                [
                    1
                ]
            ]
        }
    },
    {
        "name": "print_result",
        "start": 331,
        "end": 393,
        "inTypes": [
            "double"
        ],
        "inTypesV": [
            true
        ],
        "in": [
            "A"
        ],
        "outTypes": [],
        "outTypesV": [],
        "out": [],
        "fragmented": [
            0
        ],
        "fragmentedLoopsNest": [
            {
```

```
                    "inductives": [
                        "i"
                    ],
                    "inits": [
                        0
                    ],
                    "ends": [
                        "N"
                    ],
                    "start": 331,
                    "end": 393,
                    "body_start": 368,
                    "body_end": 393
                }
            ],
            "shadow": {}
        }
    ],
    "LuNAOperators": [
        {
            "type": "computantionalFragment",
            "CF": "init_A",
            "args": [
                "A"
            ],
            "fragmented": "nf",
            "ordered": false,
            "reduction": []
        },
        {
            "type": "loop",
            "start": 0,
            "end": "T",
            "operators": [
                {
                    "type": "computantionalFragment",
                    "CF": "blur",
                    "args": [
                        "A",
                        "A"
                    ],
                    "fragmented": "nf",
                    "ordered": "false",
                    "reduction": []
                }
            ]
        },
        {
            "type": "computantionalFragment",
            "CF": "print_result",
```

```
            "args": [
                "A"
            ],
            "fragmented": "nf",
            "ordered": "true",
            "reduction": []
        }
    ],
    "macrosCpp": [
        {
            "start": 0,
            "end": 15
        }
    ],
    "mactorFa": [
        {
            "start": 18,
            "end": 32
        }
    ]
}
```