# Обзор языка Chapel

Докладчик: Ткачёва Настя

# Введение

Chapel (Cascade High-Productivity Language)

- Разработка компании Cray
- В рамках проекта DARPA HPCS

Состояние: в разработке

Разрабатывается как open source проект

# Цель

Целью разработки языка было повысить продуктивность программиста

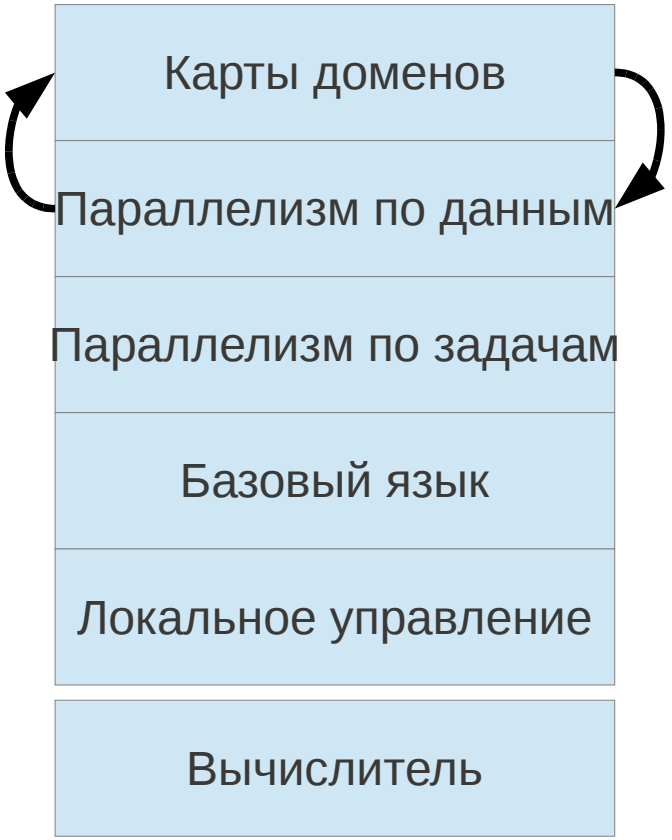
Продуктивность = Программируемость + Производительность + Переносимость + Надежность

# Методика поэтапной разработки

Поддержка нескольких уровней разработки

- Высокие уровни для програмируемости и продуктивности
- Низкие уровни для большей степени контроля

# Chapel language concepts



# «Hello World» in Chapel

```
module Hello{
proc main(){
    writeln(«Hello,world!»);
    }
}
```

# Типы данных

- Базовые типы
- Перечислимые типы
- Типы мест действий
- Структурированные типы
  - Классы
  - Записи
  - Объединения
  - Кортежи (гомогенные, гетерогенные)

# Типы данных (2)

- Типы для описания параллелизма по данным
  - Интервалы ([low]..[high])
  - Домены
  - Массивы <имя массива>[интервал] <тип элементов>

  Тип синхронизации sync

```
config const numIters = 100000;
const WorkSpace = {1..numIters} dmapped Block(…);

forall i in WorkSpace do
  writeln("Hello, world! ",
          "from iteration ", i, " of ", numIters,
          " on locale ", here.id, " of ", numLocales);
```

# Интервалы

([low]..[high])

by  -задание шага

#  - выбирает указанное число элементов из интервала

[ ] - с помощью оператора можно выполнить пересечение интервалов

# Способы управления передачей аргументов (intents)

- In
- Out
- Inout
- Const
- Param и type

# Поддержка высоко-уровневых операций

- Reductions (+,min,max,*,...)

  Ex: x =+ reduce A //sets x to sum of elements of A

- Scans

  Похоже на редукция, но высчитывает значения по порядку

  A = [1,3,2,5];

  B=+scan A; //sets B to [1,1+3=4,4+2=6,6+5=11]

# Поддержка высоко-уровневых операций (2)

B = f(A) //применение f ко всем элементам A для любого f

- Есть уже стандартные

  C =A+1;

  D = A+B;

  E = A*B;

# Домены

- Объект верхнего уровня, предназначенный для описания множества индексов
  - ключевое понятие для параллелизма по данным
  - может быть распределен между несколькими исполнителями

  Можно создавать поддомены.

For loops (последовательное исполение)

for i in domainD do ….

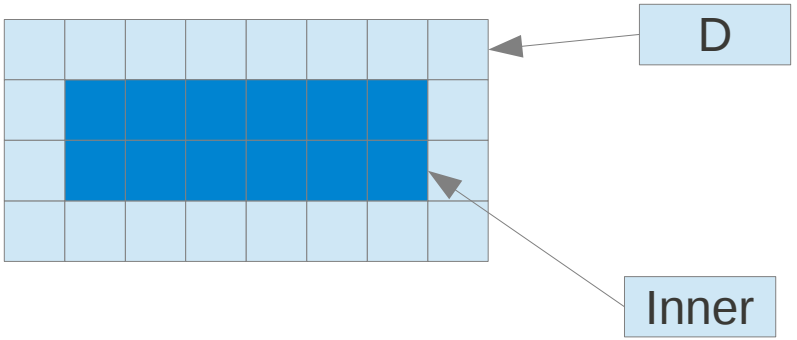
Forall loops

forall i in domainD do ….

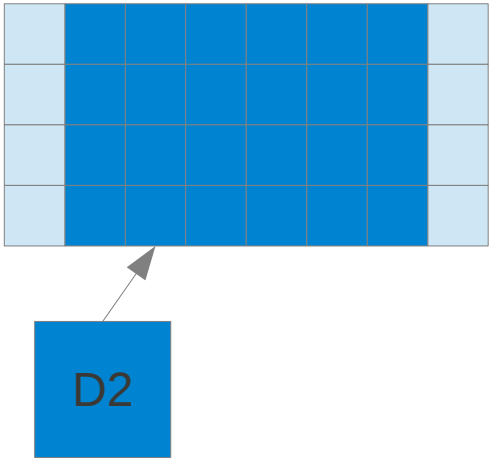
- Исполняет  параллельно каждую итерацию цикла
- Должен быть сериализумым (позволять исполнение на одном процессоре)

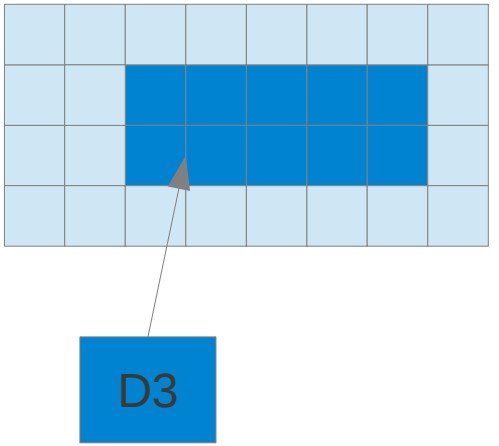
# Domain algebra

```
Config const m = 4, n = 8;
var D: domain(2) = {1..m,1..n};
var Inner: subdomain(D) = {2..m-1,2..n-1};
```



```
var D2 = Inner.expand(1,0);
var D3 = D2[D3]; //пересечение
```



# Управление параметрами параллельного запуска

--dataParTaskPerLocale=#

устанавливает количество задач, которые будут созданы для выполнения циклов forall.

--dataParIgnoreRunningTasks=[true|false]

если значение = false, то количество задач для выполнения цикла уменьшается на количество запущенных задач.

--dataParMinGranularity=#

устанавливает минимальное количество итераций, которые выполняются в рамках одной задачи.

# Переменные синхронизации

Sync [full | empty]

Пример критической секции

```
var lock1: sync bool;
lock1 = true;
critical();
var lockval = lock1;
```

# Параллелизм задач

- Задача — еденица параллелизма в Chapel.

  Создание одиночной задачи (begin):

  begin task1();

  task2();

  Создание нескольких задач (cobegin)

  cobegin{

  task1(); task2(); task3();

  }//ожидание завершения

# Цикл coforall

При использовании параллельного цикла coforall задачи создаются для выполнения всех итераций цикла. Родительская задача ожидает завершение всех дочерних задач.

# Синхронизация задач

- Атомарные операции

```
atomic{
    a+=1;
    c = a+b;
}
```

# Синхронизация подзадач

```
Sync {
begin task1();
begin task2();
begin task3();
}
```

```
cobegin{
task1();
task2();
task3();
}
```

# Ограничение числа создаваемых параллельных задач

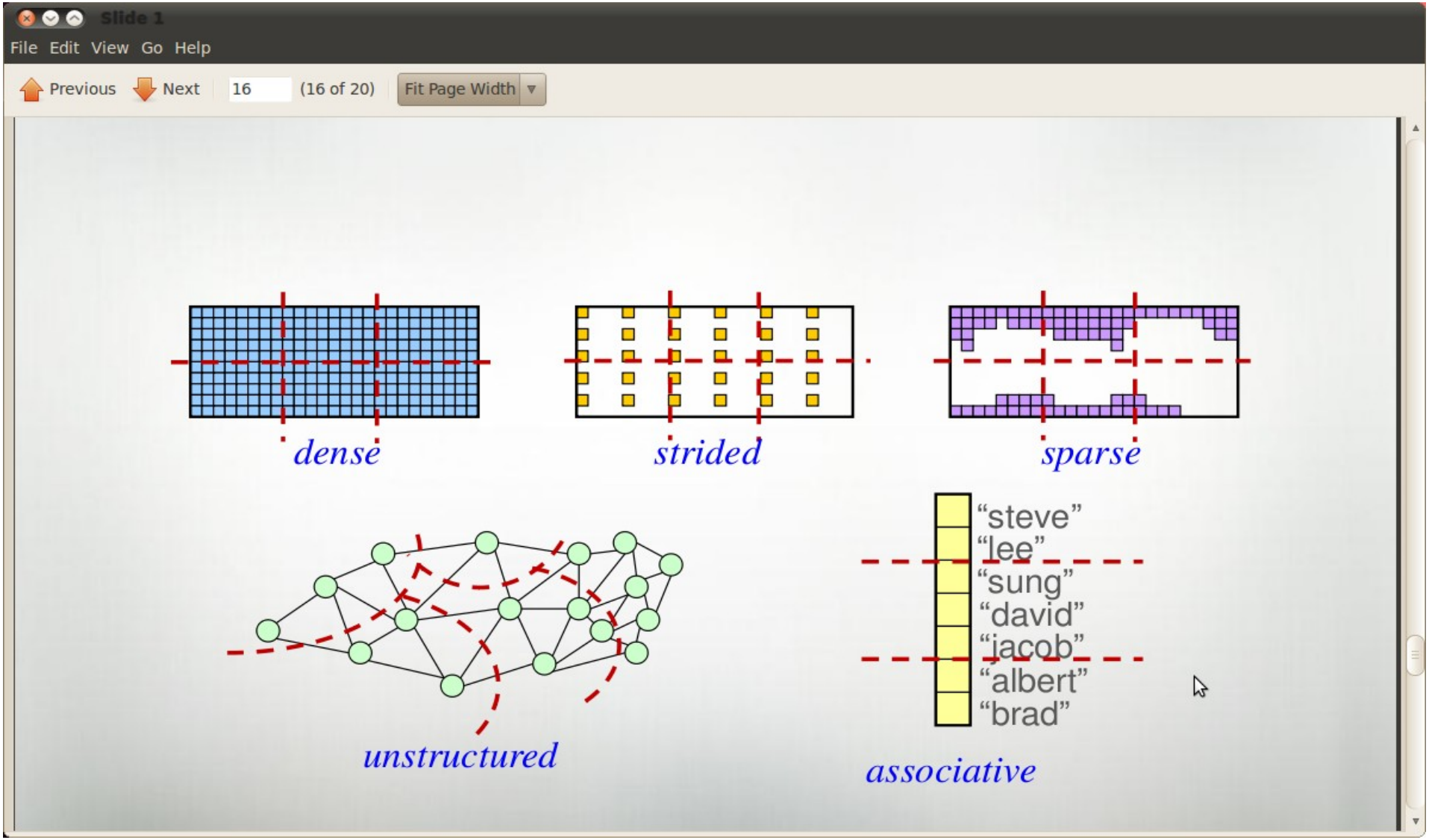
- --maxThreadsPerLocale = <i>

  максимальное число потоков, которые могут создаться на одном узле. Если равен 0, то системное значение

  Для ограничения числа создаваемых задач используется

- Serial <условие> {блок операторов}

# Нерегулярные домены



# Понятие исполнителя (locale)

Характеризует элемент параллельной архитектуры, который выполняет задачи и хранит переменные.

--numLocales = #

Методы типа locale

- Id
- Name
- NumCore
- physicalMemory

# Распределение доменов между исполнителями

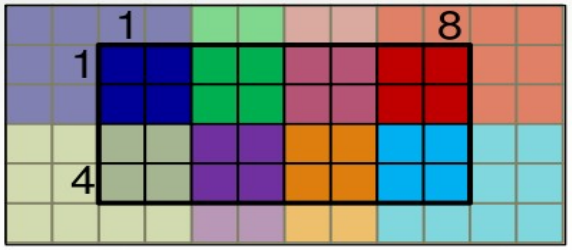
- Для описания распределения домена между исполнителями вводится понятие карта домена — это набор правил, который указывает компилятору как отобразить глобальную модель параллельных вычислений на память и процессоры исполнителя

# Категории карт доменов

- Размещения ориентированы на отдельный узел, имеющий в своем распоряжении общую для всех исполнительных элементов память (многоядерный узел).
- Распределения ориентированы на разделенные участки памяти (кластер).

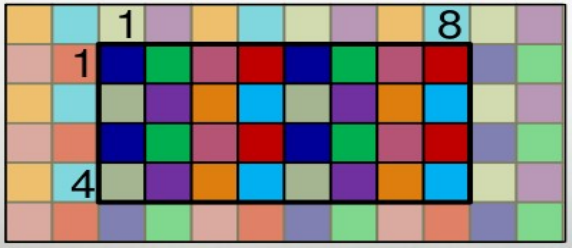
# Some Standard Distributions: Block and Cyclic

```
var Dom = {1..4, 1..8} dmapped Block(boundingBox={1..4, 1..8});
```




*distributed to*


```
var Dom = {1..4, 1..8} dmapped Cyclic(startIdx=(1,1));
```




*distributed to*

# Задание распределения

- Распределение
- Домен, определяемый на базе распределения
- Долю домена, которая приходится на одного исполнителя;
- Массив определяемый на базе домена
- Порция массива которая хранится на каждом исполнителе.

# Управление исполнителями

- Ключевое слово on на каком исполнителе исполнить следующий за ним блок

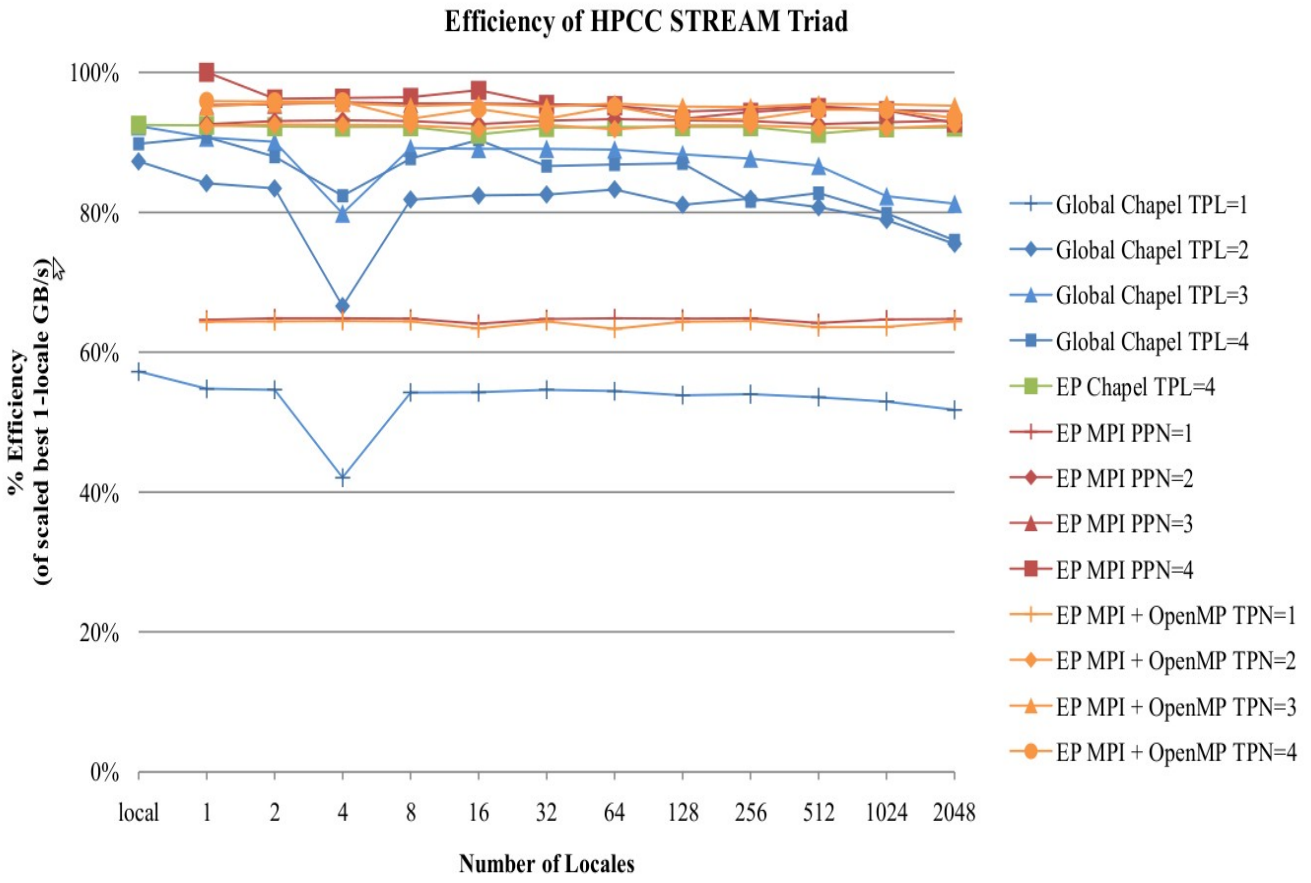
  on Locale(1)  do

  Для каждой переменной можно определить где она храниться.

# Пример локального распредления

```
Var x: int;      // x is stored on locale 0
on Locales[1] {
  var y: int;                // y is stored on locale 1
on Locales[2] {
  var z: int;                // z is stored on locale 2
on y { y-=1;}                // execute on locate 1
}
}
```

# 6 STREAM Triad Performance



# 8 Random Access Performance