СРЕДСТВА ДЛЯ ПРОГРАММИРОВАНИЯ ПАРАЛЛЕЛЬНЫХ ПРОГРАММ

Проект OST (Objects – Space – Time)

Кто работает:

1. **Два сотрудника ИПМ РАН и кафедры «Вычислительная механика» мехмат МГУ.**
2. **Аспиранты и студенты МГУ и ИПМ РАН.**

Что работает:

3-я версия на суперкомпьютерах ИПМ, счет нескольких реальных задач газовой динамики.

- OST – это развитие проекта УДВ (O-O RPC), ИПМ РАН, 80-90 гг., сейчас порядка 30 тысяч терминальных станций в Сбербанке.
- Парадокс простоты:

  1. Множество объектов, взаимодействующих путем вызова операций друг в друге.

«Прозрачность» сети

  2. Связывание путем передачи ссылок на объекты как параметров вызовов.

Неприятие «сетевиками»

- OST – естественное расширение О-О RPC

Недостатки RPC **Что добавил OST**

1.Ручное -> Автоматическое связывание через **«формальных - фактических соседей»** плюс **«топология связей»**

2.Ручное -> автоматическая **подкачка** распределение объектов по процессорам

3.Ручная ->Автоматическая синхронизация по **локальным временам** объектов

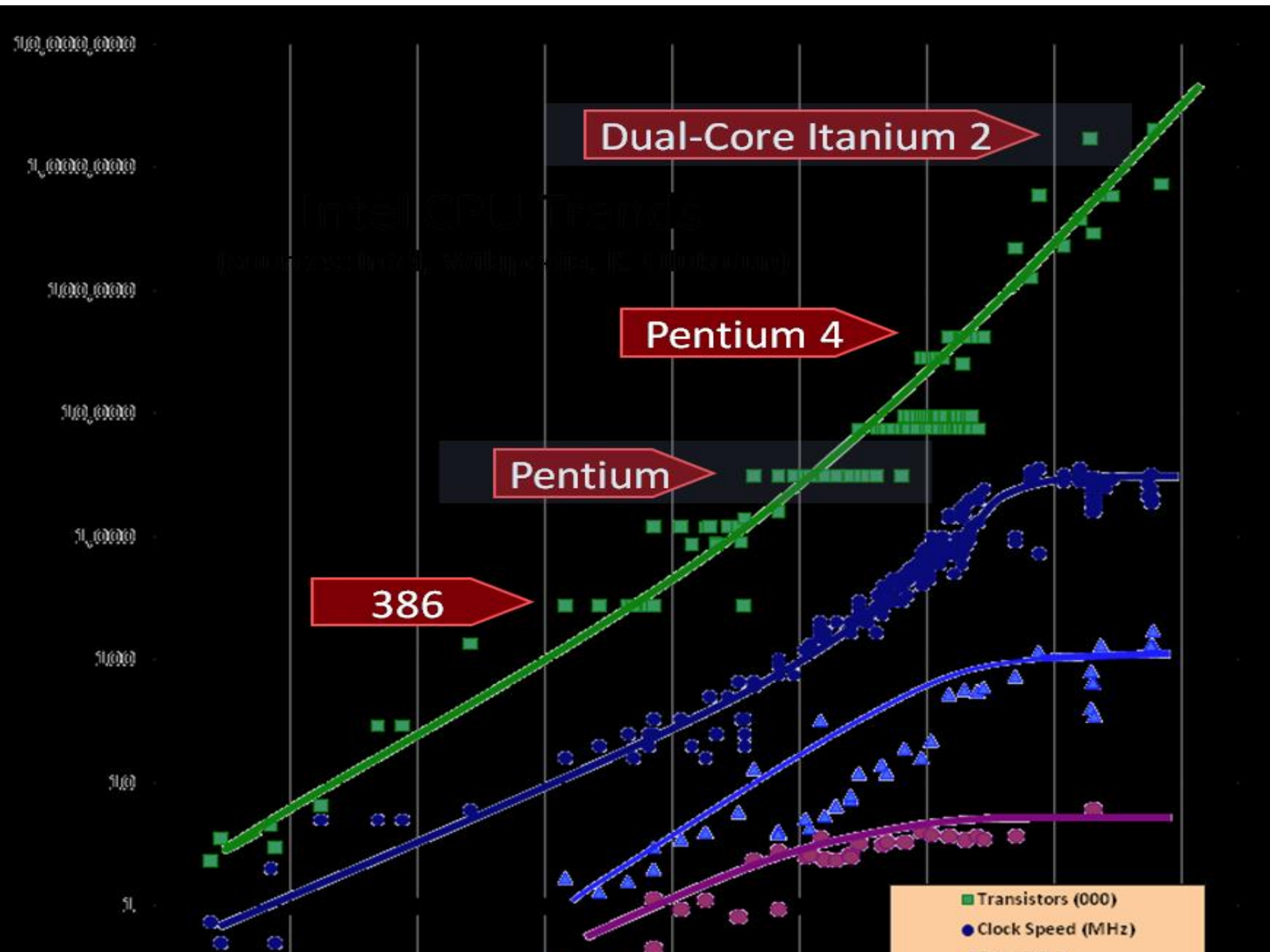
Основной результат

- Локальность программирования объекта в «окружении формальных соседей»
- Сборка и эволюция параллельной системы – автоматически
- **Традиционное распараллеливание / предлагается композиция**

Главная проблема: Джон Хэннеси, президент Стэнфордского университета –

“… когда мы начинаем говорить о параллелизме и легкости использования действительно параллельных компьютеров, мы говорим о проблеме, которая труднее любой проблемы, с которой встречалась наука о компьютерах … Я бы запаниковал, если бы я работал в промышленности.”

Главная цель – свести трудоемкость создания **параллельных** моделей к трудоемкости создания **последовательных** моделей для максимально широкого класса задач.


Dual-Core Itanium 2
Pentium 4
Pentium
386
Transistors (000)
Clock Speed (MHz)

ИСПОЛЬЗОВАННАЯ МОДЕЛЬ ПРОГРАММИРОВАНИЯ

Множество объектов, взаимодействующих путем вызова операций друг в друге,
плюс **множество процессов**.
Соотношение конструкций «**объект-процесс**» можно охарактеризовать как «**много-много**». Это соотношение означает, что один процесс может пройти через много объектов (в том числе расположенных **на разных процессорах**), а в один объект могут войти и одновременно существовать много процессов.

Что добавлено:

1. Эволюция объектов в **пространстве и времени**.

2.Топология связей объектов на основе **координат**.

3.Локальное **время** объекта.

Общее для любых систем из частей:

- топология связей
- согласованная эволюция частей

Именно это и автоматизируется.

МОТИВЫ ВЫБОРА ПРЕДЛАГАЕМЫХ СРЕДСТВ

Уровни представления физической области:

Физическая модель -> Математическая модель -> Вычислительная модель -> Программная модель -> МВС

Принцип **подобия моделей . Топологическое** подобие.

Проблема, возникшая при построении модели на одном уровне представления, обязательно имеет аналог на других уровнях представления. При выборе варианта решения очень часто целесообразно рассмотреть, как решается эта проблема на других уровнях представления. Другая формулировка этого правила — нужно видеть **физический смысл** действий с вычислительными моделями.

КОКРЕТНЫЕ РЕШАЕМЫЕ ПРОБЛЕМЫ

- **Управление связями** между объектами
- **Синхронизация** эволюции объектов

Прикладной программист **OST**

1.Локальное описание: 1. Построение связей

- Окружение – список **фактические соседи**

формальных соседей 2. Синхронизация

- Локальный алгоритм 3. Подкачки объектов

- Локальное время 4. Файл объектов

2.Топология связей 5. Средства отладки

```
# Структура класса объекта
class applied_object(ost.Object.Abstract):
    # Класс прикладного объекта
    def fun_1(self, ...):
        # Содержимое функции
    def fun_N(self, ...):
        # Содержимое функции

        # Функция начала вычислений
    def run(self):
        # self - аналог this в c++
        self.setFinish()
#-----------------------------------------------#
```

def run(self):

 # self.topology.neighbors список формальных соседей

#Обращение к i-ому соседу по fun_j-ой функции

self.topology.neighbors[*i*].**link**.*fun_j*()

При наличии синонимов использование проще

Обращение к соседу слева по fun_j-ой функции

self.*left*.*fun_j*()

Вызов произойдет только при **равенстве локальных времен** вызывающего и вызываемого объектов

После окончания вычислений завершаем вычисления

self.setFinish()

В переменной **self.time** хранится текущее время объекта для синхронизации

Для продвижение на шаг *time_step* делается запрос к монитору OST

возврат из которого произойдет только после продвижения

self.setXYZT(**self.time** + *time_step*)

Объекты могут менять свое положение в пространстве

Получение актуального массива координат текущего объекта

coord = **self.topology.get_coordinates**()

Изменение *coord*

Монитор OST изменяет координаты

только вместе с продвижением по времени

self.setXYZT(**self.time** + *time_step*, *coord*)

Примеры в небесной механике и молекулярной динамике

```
# Функция задает интерфейс из формальных соседей
# Задает синонимы и интерфейсы данного соседа
    def init_topology(self):
    # Задаем интерфейс и синоним для формального соседа i1
        self.init_neighbor(i1, Class_interface_i1, "synonym_i1")
...
     # Задаем интерфейс и синоним для формального соседа iK
        self.init_neighbor(iK, Class_interface_iK, "synonym_iK")
```

```
# Структура класса с функцией описания окрестности
class applied_neighbr_topol(ost.Topology.Abstract):
# Функция описания окрестности точки в пространстве
    def neighborhood(self, p):
# p - набор(массив) координат, рассматриваемой точки
# Возвращает массив(словарь) наборов(массивов)
координат точек, которые входят в окрестность
рассматриваемой точки p
        # с заданием синонимов "соседей"
        return { synonym1: [p1_x1,...,p1_xn1],  ....,
                 synonymk: [pk_x1,...,pk_xnk]
               }
```

```
# Пример топологии Графа на основе функции описания окрестности
class graph_neighbr_topol(ost.Topology.Abstract):
    def __init__(self):
        #Таблица, задающая ребра графа
        self.edges = { 0 : [1, 3], 1 : [0, 2, 3],
                       2 : [], 3 : [2]
                     }
    # Функция описания окрестности вершины p
    # Возвращает список всех вершин, с которыми соединена данная p
    def neighborhood(self, p):
        return self.edges[p]
```

```
class radius_topology(ost.Topology.Abstract):
    def __init__(self):
        # Задание радиуса окрестности точки
        self.radius = 2
#Функция вычисления расстояния между точками
def distance(self, p1, p2):
return sqrt( (p1[0] - p2[0])**2 +(p1[1] - p2[1])**2)
    # Функция близости. Растояние между точками меньше, чем radius
    def proximity(self, p1, p2):
        return self.distance(p1, p2) < self.radius
```

```
class line_topology(ost.Topology.Abstract):
#Функция описания окрестности вершины p
    # В случае прямой справа и слева
    # left и right синонимы
    def neighborhood(self, p):
        return { left : p[0] - 1,
                 right: p[0] + 1
               }
```

Программа инициализации

Начинаем инициализацию модели

Создаем объект инициализации.

Сохраняем в файле modelname.mod

obj_init = **ost.Core.Init**("*modelname.mod*")

Задаем класс глобальной топологии

В данном случае кольцо с 10 элементами

obj_init.**topology** = ***ost.Topology.Ring(N** = 10)*

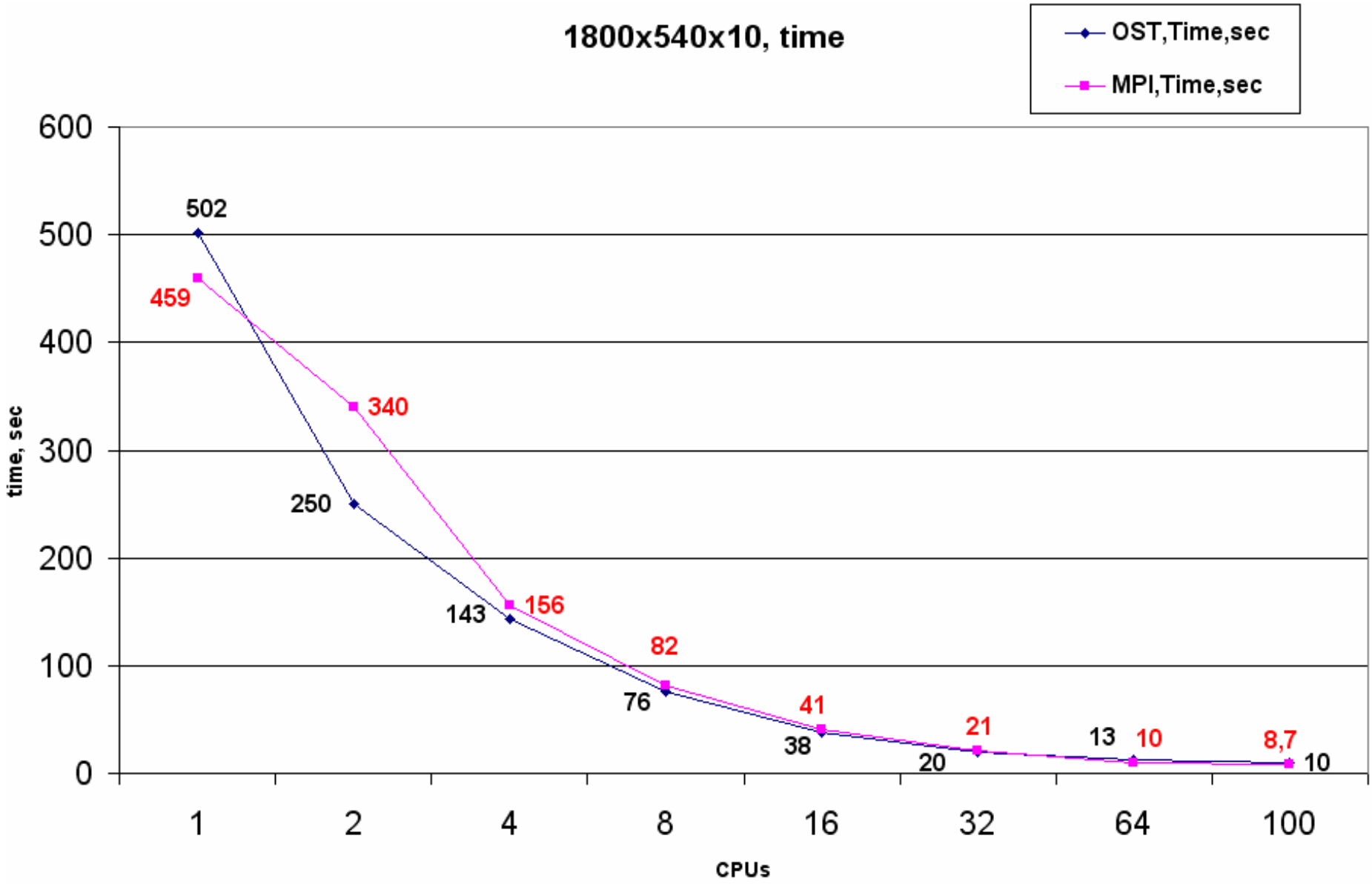
```
# цикл, в котором создаются объекты модели
# В примере создаются 10 объектов, для кольца
for index in xrange(0,10):
   # Создание объекта типа applied_object
app_object = obj_init.create_object( applied_object )
# объект заполняется необходимыми данными
# можно задать локальную топологию объекта
# В данном примере мы описываем окрестность
состоящую из элементов P1,…,Pk
app_object.topology =
ost.topology.Neighborhood([P1,…,Pk])
   # Помещаем объект в точку пространства объектов
   # Точка имеет координату index
   obj_init.topology.set(app_object, index)
```

Добавление файла с программным кодом объекта
С помощью аналогичных конструкций можно добавить и другие данные
Цель – самоопределенный и самодокументируемый файл с моделью
obj_init.**addSorceFile**("*applied_objects.py*")
Сохранение модели в файле
obj_init.**save**()

Что конструктивно новое

- Формализация «окружения» объекта в виде списка формальных соседей.
- Автоматическое построение связей между объектами и замена формальных соседей фактическими по топологии, заданной пользователем.
- Актуализация ссылок на фактических соседей при совпадении локальных времен.

Результаты счета, оценка эффективности



MPI vs. OST

- синхронизация вычислений полностью ложится на плечи прикл программиста;
- все межпроцессные связи полностью организует прикл программист;
- модель процессов; удаленное обращение к соседям – через рассылку сообщений:

```
MPI_Isend(data,Numb,MPI_DOUB
LE, IND,0,MPI_COMM_WORLD,
&sendreq)
```

- предоставляет автоматический механизм синхронизации вычислений:

```
self.setXYZT(self.time+1)
```

- автоматическое назначение ссылок на соседей по заданной топологии;
- объектная модель; удаленное обращение к соседям – как локальный вызов:

```
self.left.somefun(x)
```

Технические характеристики

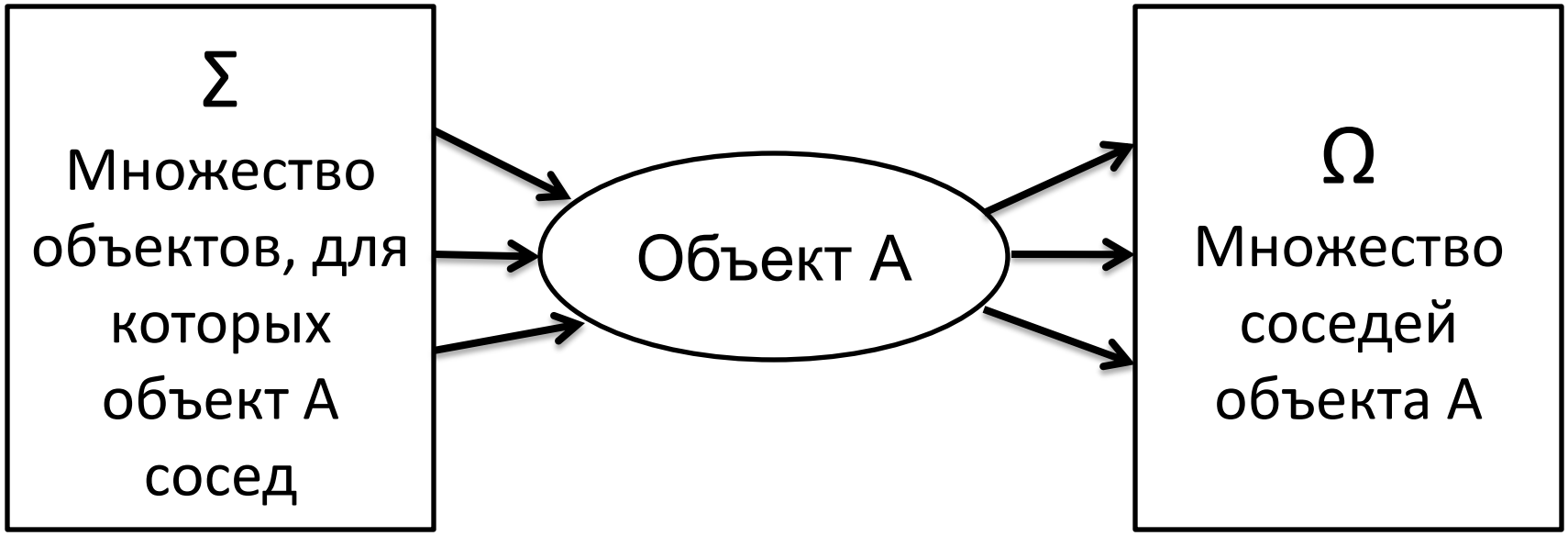
- 1. Коэффициент эффективности параллельного счета Кр > 0.9
- 2. Практически автоматическая сборка (установление связей, синхронизация взаимодействия по локальным временам объектов) автономно запрограммированных объектов (моделей физических подобластей). То-есть, трудоемкость параллельного программирования сводится к трудоемкости раздельного программированию последовательных алгоритмов для подобластей.
- 3. **Счетная часть объектов – C++, Фортран, управляющая часть Python.**
- 4. Средства отладки – визуализация, отладчик, профилирование и т.д.

- 5. Хранения множества объектов программной модели в файле объектов (базе данных). Контрольные точки и рестарты. Мобильность хранимой модели – возможность перенести модель на другую вычислительную установку после окончания очередного этапа счета и продолжить его на новом месте. Пример, короткий отладочный счет на PC и продолжение на МВС.
- 6. Планировщик ресурсов, обеспечивающий автоматическую подкачку/выталкивание программных объектов между процессорами и файлом объектов в стиле подкачки страниц в операционных системах. Счет на основе “рабочего множества” объектов.

Заключение

- Суть предлагаемого решения – локальность всех описаний: связей, времени и алгоритмов эволюции подобластей. Модель всей области получаем практически автоматически путем композиции подобластей по топологии, определенной программистом.
- Дополнительная информация на сайте ost.kiam.ru
- Перспективы - разнонаправленные

Алгоритм синхронизации

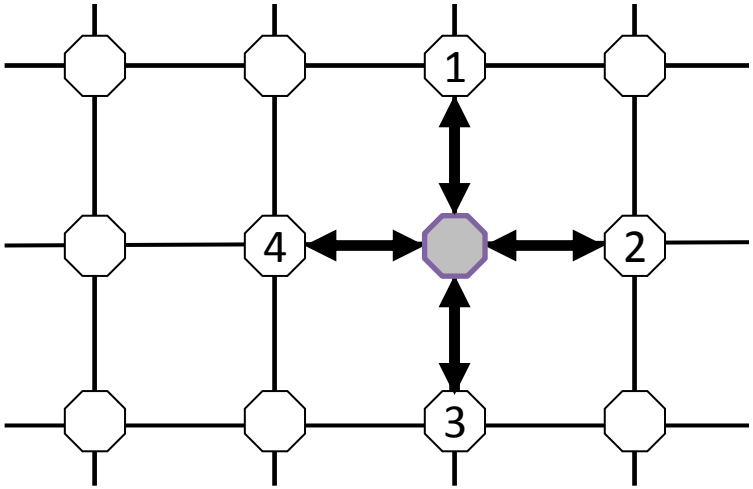


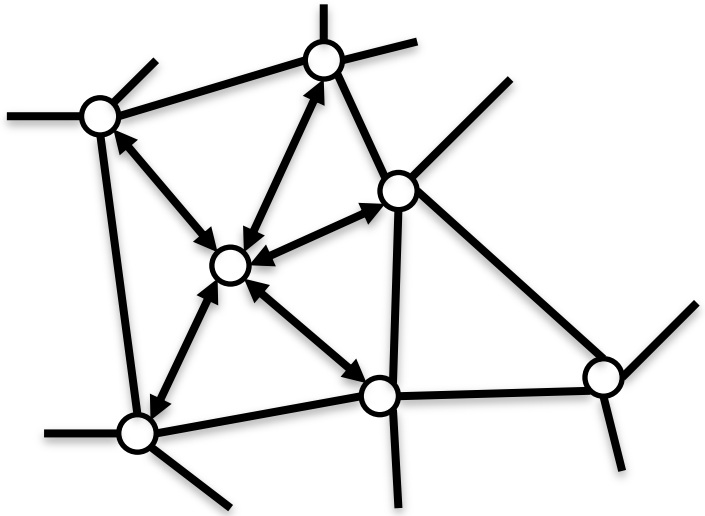
1) Объект А запрашивает продвижение от t к t+1
2) Продвижение разрешается, когда все объекты в Σ достигнут времени t
3) Запросы из объекта А в объекты из Ω только при равенстве времен.

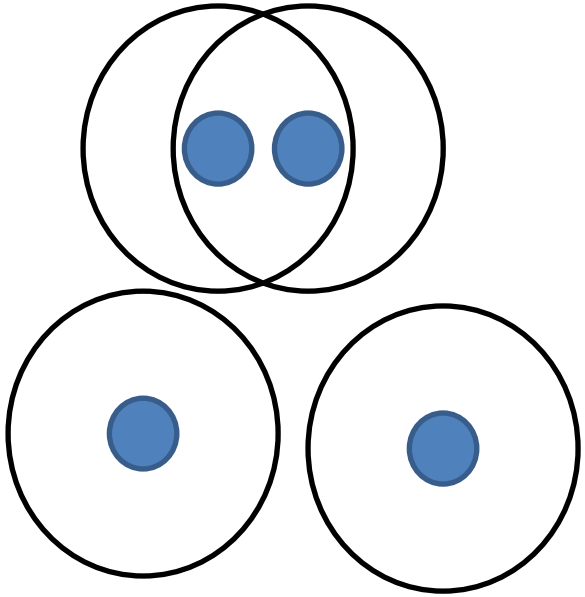
Топология связей между объектами.

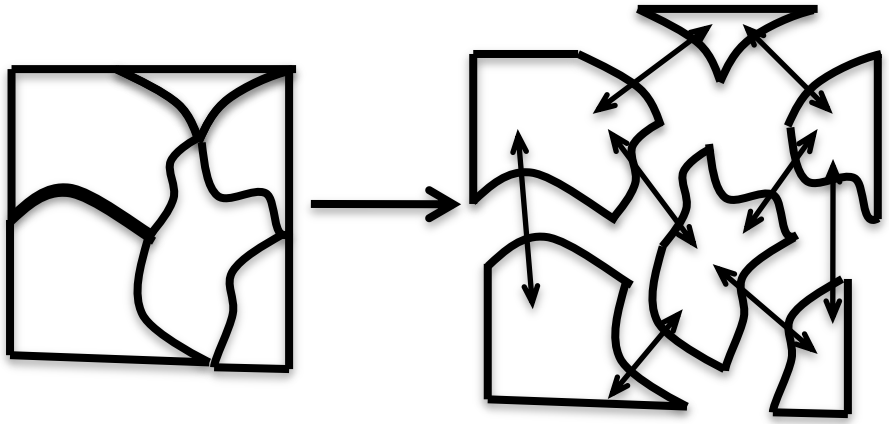
- Определение связей между объектами через локальное описание окрестности для каждого объекта.
- Автоматическая замена формальных соседей на фактические во время счета.

Примеры связей





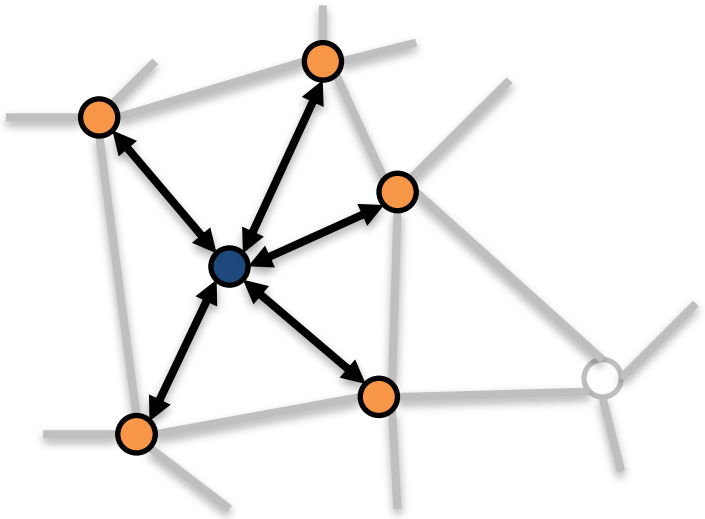




Окрестность узла

•Окрестность - узлы, с которыми соединен данный узел.

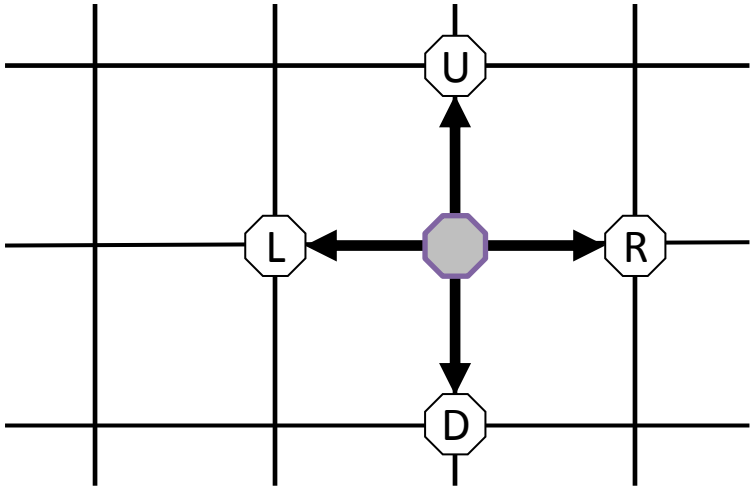
•Узлы можно отожествить с вычислительными объектами



•Окрестность может изменяться по ходу вычислений.

Пример: целочисленная решетка

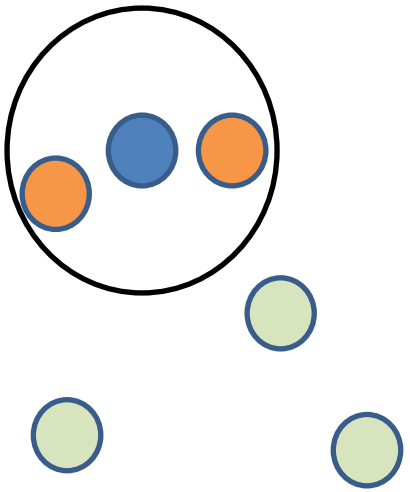
Координаты соседей отличаются на ±1 по одной из координат



Пример: плоскость

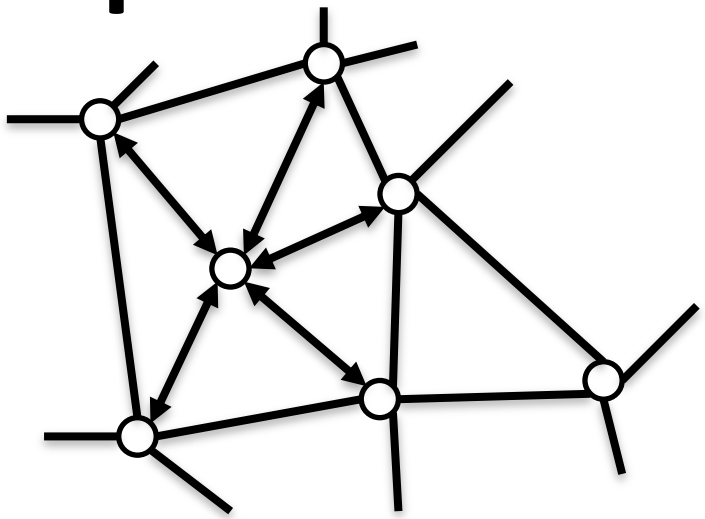
Соседи удалены не более, чем на **r:** $|x - y| \leq r$

Функция близости проверяет критерий попадания в окрестность (круг)

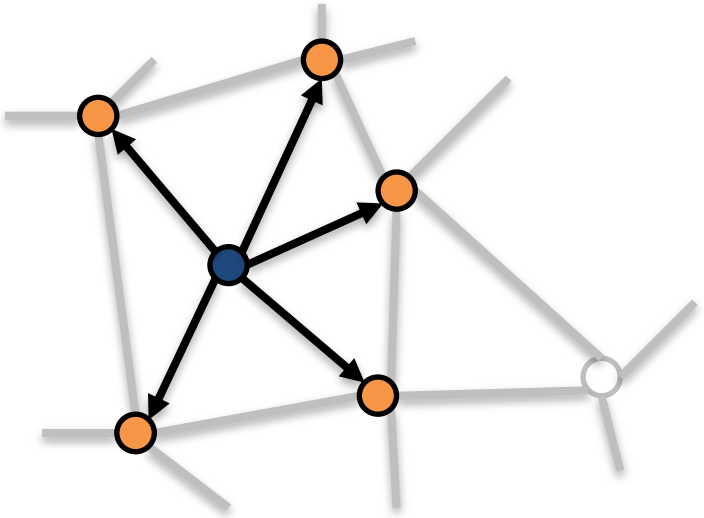


Пример: неструктурированная сетка

Все вершины графа пронумерованы 1,2,3,…



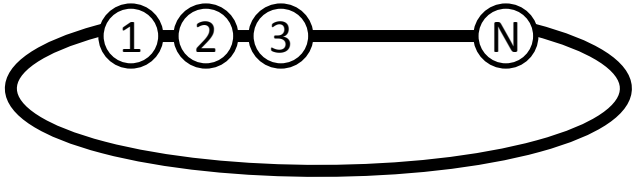
Для каждого узла явное описание списка соседей



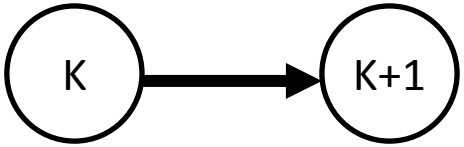
Простое использование формата METIS для такого описания

Пример: кольцо

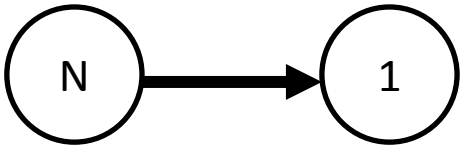
•Кольцо объединяет в себе 2 вида локальных топологий



• N – 1 одинаковых одномерных



• 1 вырожденная, замыкающая кольцо



Базовая топология – связный граф

- Случай неструктурированной сетки

```
#Создание объектов в функции инициирования модели
for <…>:
    myobject = objInit.createObject(objectClass)

<Заполнение данными объекта myobject >
    object.array1 = <…> …
<Передача базового объекта topology в object>
myobject.topology = topologyLocal()
<Занесение номера объекта в глобальный каталог>
    objInit.topology.set(myobject, index)

# Задание окрестности для каждого объекта
for <…>:
<Определение списка соседей>
    myobject.topology.set(
                index, [nei_index1,…,nei_indexN] )
```