О парадигме универсального языка параллельного программирования

А.В. Климов
arkady.klimov@gmail.com

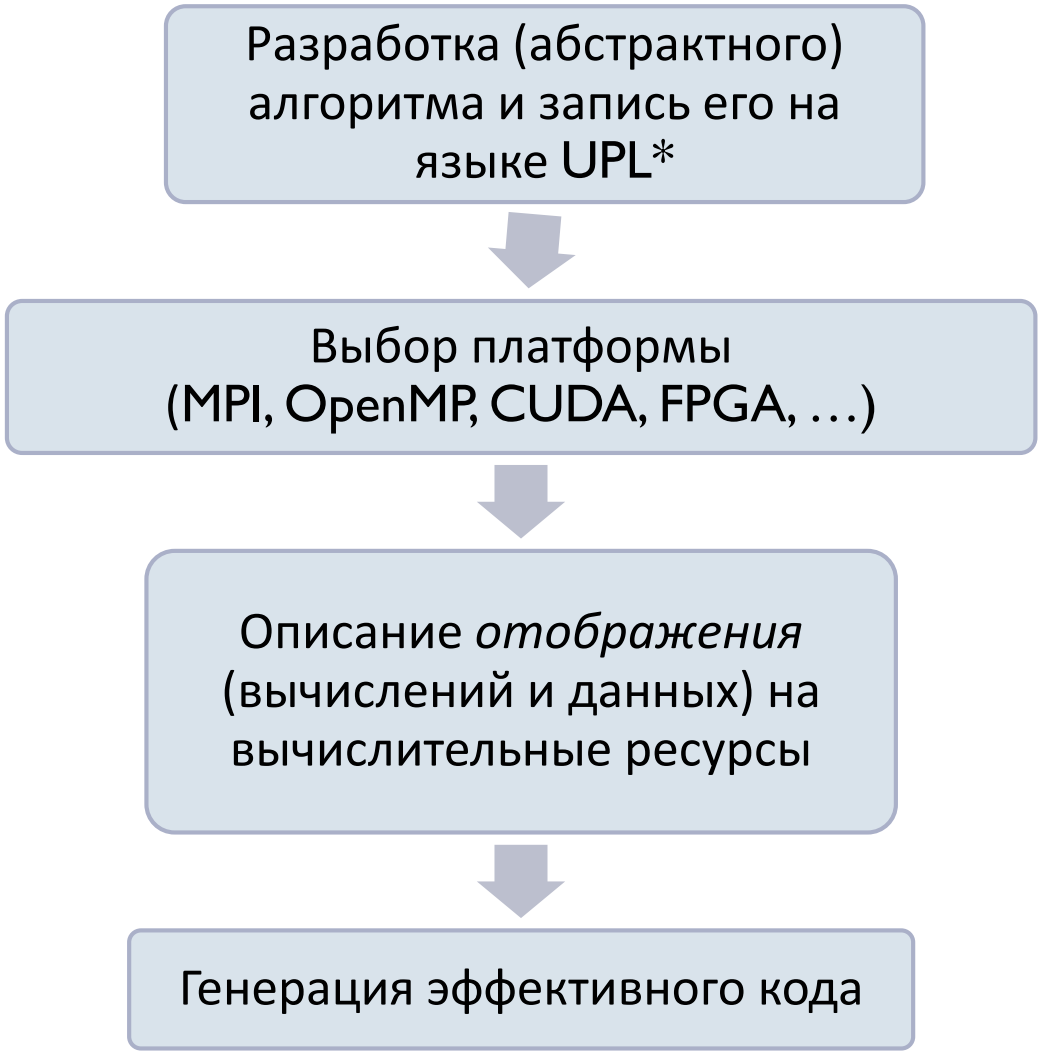
Институт проблем проектирования в микроэлектронике (ИППМ) РАН

Языки программирования и компиляторы
PLC’2017
Ростов-на-Дону
3 апреля 2017

Проблема параллельного программирвоания

- Параллельное программирование – трудное дело
- Существует множество разных параллельных архитектур и платформ
- Возможно ли иметь один язык для записи *абстрактных* (минимально зависящих от свойств «железа») алгоритмов и компилировать с него для всех платформ?
- Это парадигма WORE (или WOCA):
  Write-Once-Run-Everywhere (-Compute-Anywhere)
- Должно допускаться непосредственное исполнение (быть может неэффективное)
- Возможны дополнительные средства отображения
- Какие есть еще предложения на эту тему?

Стадии разработки – что хотелось бы в идеале



*UPL – Universal Parallel Language

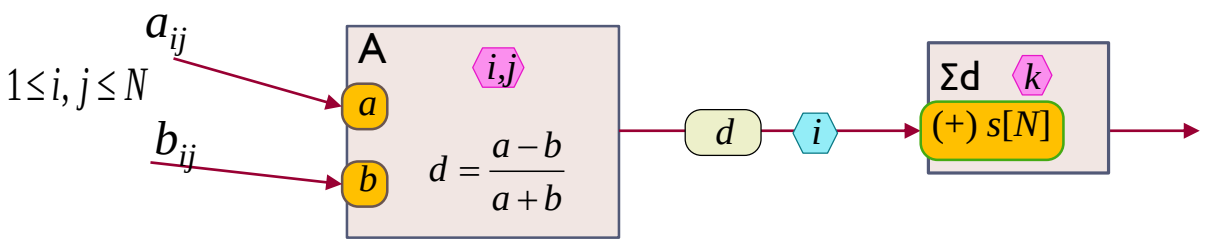
Существующие WORE проекты

(все основаны на потоковой модели вычислений)

Имя (кто, где, когда) I	Особенности	G/S	Применение
LabVIEW (National Instruments, USA, 1986)	Графический язык программирования. Есть компиляция для FPGA.	G	Ограниченная область (измерения и управление)
Single Assignment C (SAC) (University of Kiel, Germany, 1994)	Функциональный язык, синтаксис как C, строгая типизация, вывод типов, shape-инвариантность, в стиле map/fold.	G	Работа с плотными массивами
Concurrent Collections (CnC) и его производные: DFGR, PIPES. Изначально: Tstreams (HP Labs)	Двудольный граф с явными тегами. Данные и операции отдельно. Отдельные средства отображения.	G	Вычисления с детерминированным деревом задач. Парадигма «сбора».
TensorFlow (Google)	Библиотека операций над плотными многомерными массивами – тензорами	G	Deep Neural Networks
Полиэдральная модель для линейных вложенных циклов	Входной язык – обычный C или фортран (аффинное подмножество). Средств отображения нет .	G	Автоматическое распараллеливание (для аффинных программ) на OpenM CUDA. Распределение автоматическ
ПИФАГОР (Красноярский тех. Университет, Россия) Легалов и Ко. С 1995.	Оригинальный функционально-потоковый язык. Операции над векторами (плотными). Неявный if . Задержанный список. Рекурсия.	G	Исследования. От мелкозернистого параллелизма к архитектурно-зависимому коду.
UPL(G) (Москва, Россия, ИППМ РАН, 2016)	Динамический потоковый язык с явными индексами. Графический язык. Отдельные средства отображения	S	Распределенная обработка с нерегулярными и разреженными данными
Charm++ plus Charisma	Перемещаемые объекты (агенты) с состояниями, односторонние вызовы, Charisma – orchestrating language	S	То же. RTS своя (не WORE).

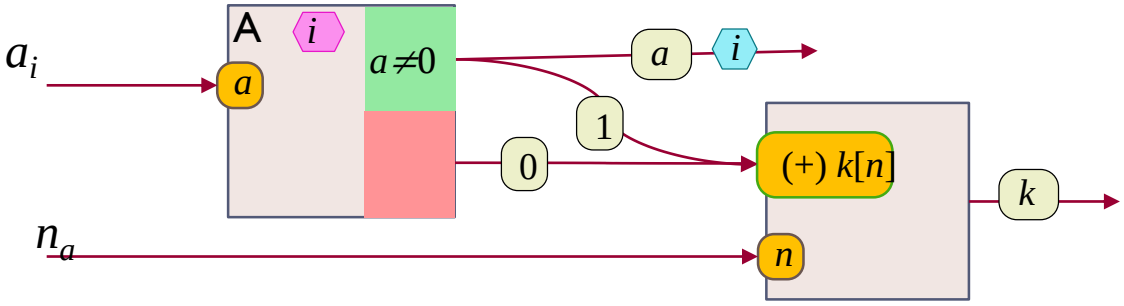
G – парадигма сбора (gather) S – парадигма раздачи (scatter)

Элементы графического языка - и их текстовые аналоги



Пример 0.1: поэлементная обработка пары матриц A и B с последующим суммированием по строкам

```
vconst N;
node A(float a,
       float b)⟨i,j⟩ {
    d = (a-b)/(a+b);
    d ⟶ SD⟨i⟩;
}
node SD(
  float reduce(+) s[N]
  ) ⟨i⟩ {
    s ⟶ …
}
```



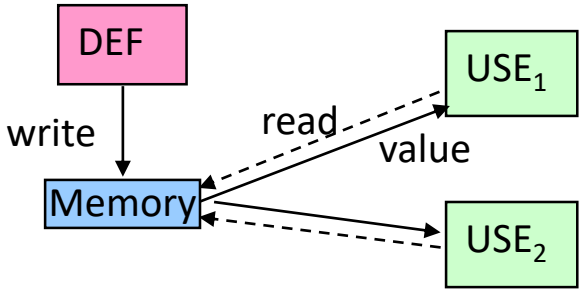
Пример 0.2: прореживание нулей и подсчет количества оставшихся элементов

```
node A(float a)⟨i⟩ {
  if (a/=0) {
    a ⟶ …
    1⟶S1.k;
  } else
    0⟶S1.k;
}
node S1(
  int reduce(+) k[n],
  int n) ⟨i⟩ {
    s ⟶ …
}
```

Парадигмы сбора и раздачи

Все традиционные языки основаны на парадигме сбора

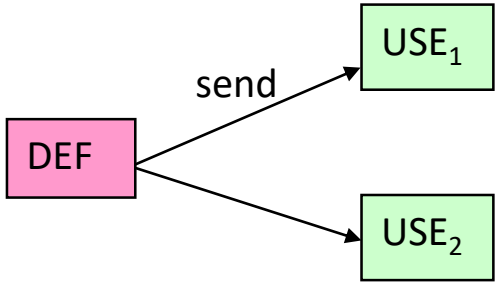
Парадигма СБОРА
(C, Fortran, etc)



Поставщик сохраняет свой результат в памяти по некоторому адресу, откуда его запрашивают потребители по мере надобности (по тому же адресу)

Обычно результат сохраняется рядом с производителем – принцип owner computes

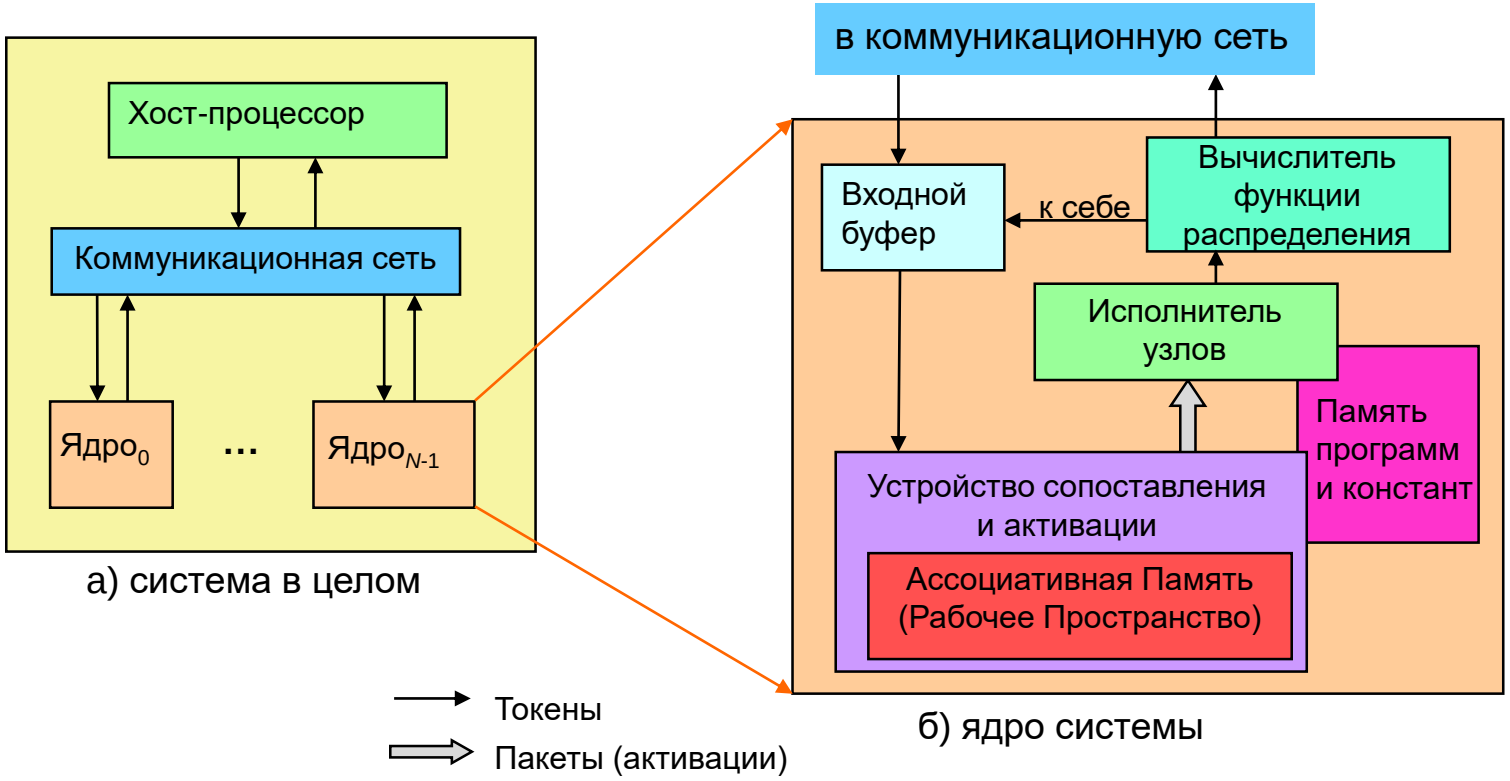
Парадигма РАЗДАЧИ
(dataflow, message passing, CHARM++)



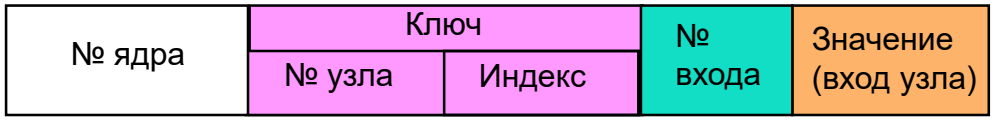
Поставщик знает (вычисляет) сам «адреса» всех потребителей, на которые и рассылает свой результат.

Для распределенного вычисления парадигма раздачи лучше:
одна передача вместо двух, нет ожидания при чтении

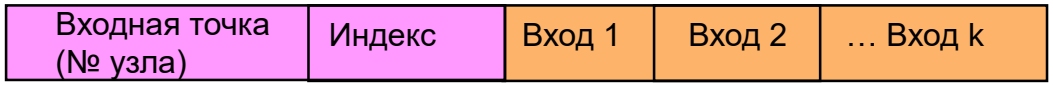
Абстрактная схема (модель) параллельного вычислителя для языка UPL



Формат токена:



Формат пакета:



Функции распределения:

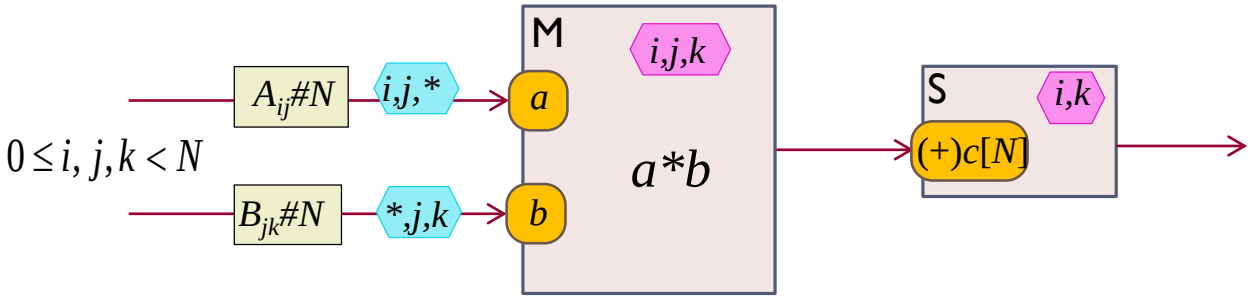
place: *Ключ* ⟶ *Номер ядра*
stage: *Ключ* ⟶ *Номер этапа*

Распределение вычислений по пространству и времени

- Распределение узлов задается функцией, зависящей от тегов (индексов) и только от них (своей для каждого типа узла).
- Функция place(tag) выдает номер процессора (ядра)
- Функция stage(tag) выдает номер этапа
- ФР применима как к исполняемому узлу, так и к токену
- Монотонность stage
- При выборе place две задачи – улучшить баланс загрузки и уменьшить число пересылаемых токенов (пространственная локальность)
- При выборе stage задача – уменьшить число откачек-подкачек, или сократить объем потребной ассоциативной памяти ключей (временная локальность)
- Значения ФР могут быть многомерными $(i,j,k,\ldots)$: для place это координаты в кубе (торе), для stage это задает упорядочение через лексикографический порядок.
- Для токенов с * ФР может порождать мультикастинг (place) или многократные «подкачки» (stage)
- Выбор ФР не влияет на функциональность (если соблюдать ряд ограничений)
- ФР – необходимый элемент при выводе эффективного кода для заданной платформы

NxN Matrix Multiplication

Основное определение алгоритма:



Текстовая форма:

```
vconst N;
node A(float a,float b)⟨i,j,k⟩ {
    a*b ⟶ S.c⟨i,k⟩;
}
node S(float reduce(+)c[N])⟨i,k⟩{
    c ⟶ …
}
```

Засылка данных:

(a_{ij})#N ⟶ V.a⟨i,j,*⟩

(b_{jk})#N ⟶ V.a⟨*,j,k⟩

Функции распределения:

$Place = (0)$

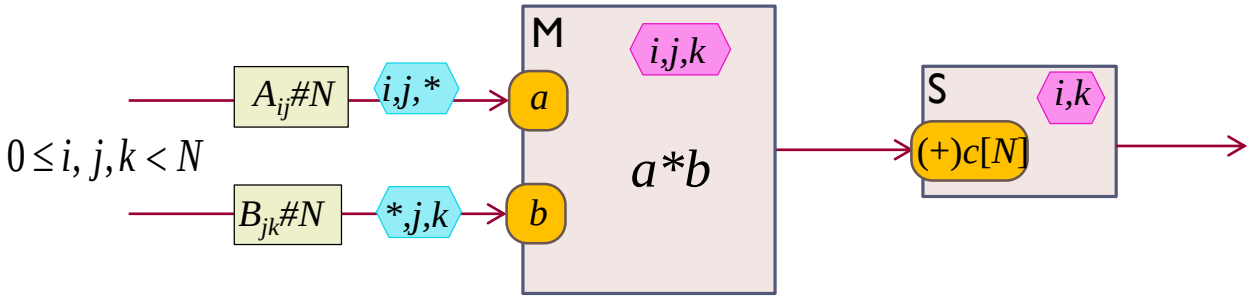
$Stage = (i,k,j)$

Стандартный последовательный алгоритм (C):

```
for (i=0; i<N; i++)
 for (k=0; k<N; k++) {
   c[i][k]=0.0
   for (j=0; j<N; j++)
     c[i][k]+=a[i][j]*b[j][k]
   }
```

NxN Matrix Multiplication

Основное определение алгоритма:



Текстовая форма:

```
vconst N;
node A(float a,float b)⟨i,j,k⟩ {
    a*b ⟶ S.c⟨i,k⟩;
}
node S(float reduce(+)c[N])⟨i,k⟩{
    c ⟶ …
}
```

Засылка данных:

$(a_{ij})\#N \longrightarrow V.a\langle i,j,* \rangle$

$(b_{jk})\#N \longrightarrow V.a\langle *,j,k \rangle$

Функции распределения:

$Place = (0)$

$Stage = (i/m, k/m, j/m, i\%m, k\%m, j\%m)$

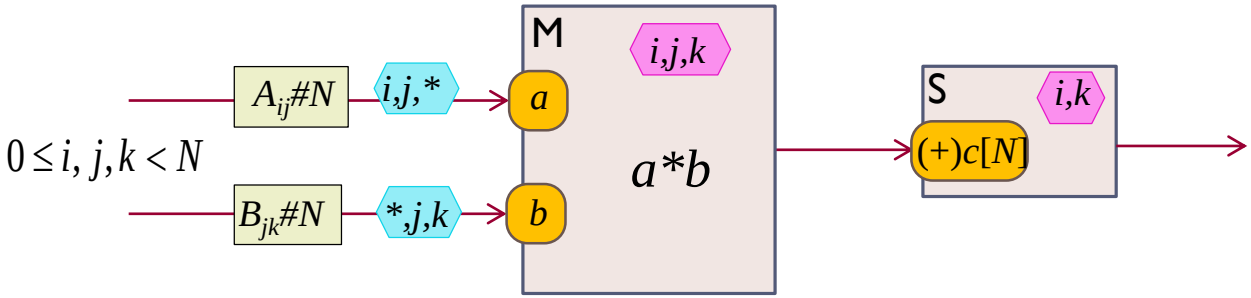
m – размер блока

Блочный последовательный алгоритм:

```
for (i=0; i<N; i+=m)
 for (k=0; k<N; k+=m) {
  for (i1=0; i1<min(m,N-i*m); i1++)
   for (k1=0; k1<min(m,N-k*m); k1++)
    c[i+i1][k+k1]=0.0
  for (j=0; j<N; j+=m) {
   for (i1=0; i1<min(m,N-i*m); i1++)
    for (k1=0; k1<min(m,N-k*m); k1++)
     for (j1=0; j1<min(m,N-j*m); j1++)
      c[i+i1][k+k1] += a[i+i1][j+j1]*b[j+j1][k+k1]
  };
```

NxN Matrix Multiplication

Основное определение алгоритма:



Текстовая форма:

```
vconst N;
node A(float a,float b)⟨i,j,k⟩ {
    a*b ⟶ S.c⟨i,k⟩;
}
node S(float reduce(+)c[N])⟨i,k⟩{
    c ⟶ …
}
```

Засылка данных:

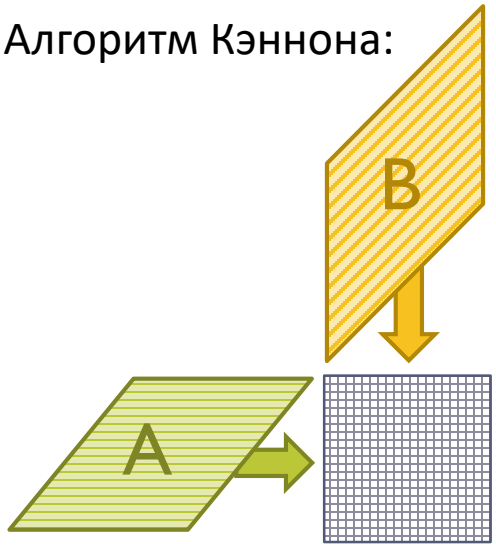
$(a_{ij})\#N \longrightarrow V.a\langle i,j,* \rangle$

$(b_{jk})\#N \longrightarrow V.a\langle *,j,k \rangle$

Функции распределения:

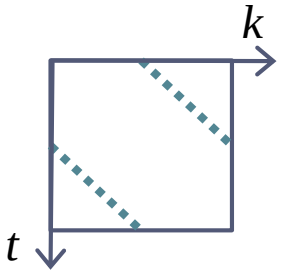
$$Place = (i,k)$$
$$Stage = t = (i+j+k)\%N$$

Алгоритм Кэннона:



На каждом такте матрицы A и B продвигаются на один элемент вправо и вниз соответственно

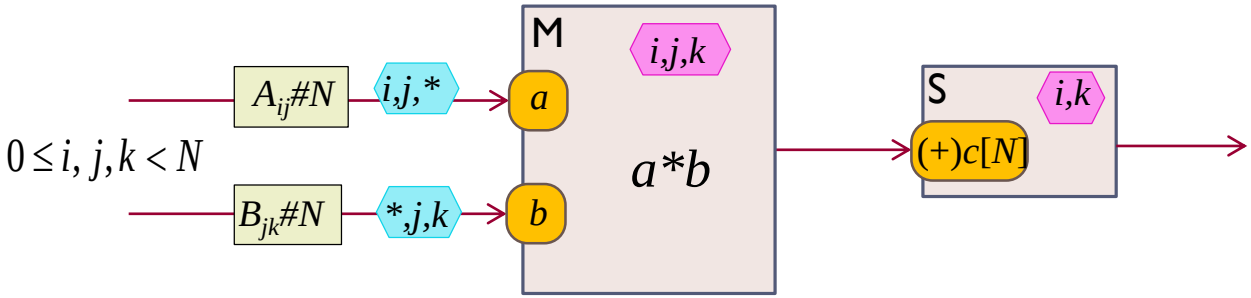
Движение элемента A_{ij} в плоскости (k,t)



Матрица процессоров (2D-тор)

NxN Matrix Multiplication

Основное определение алгоритма:



Текстовая форма:

```
vconst N;
node A(float a,float b)⟨i,j,k⟩ {
    a*b ⟶ S.c⟨i,k⟩;
}
node S(float reduce(+)c[N])⟨i,k⟩{
    c ⟶ …
}
```

Засылка данных:

(a_{ij})#N ⟶ V.a⟨i,j,*⟩

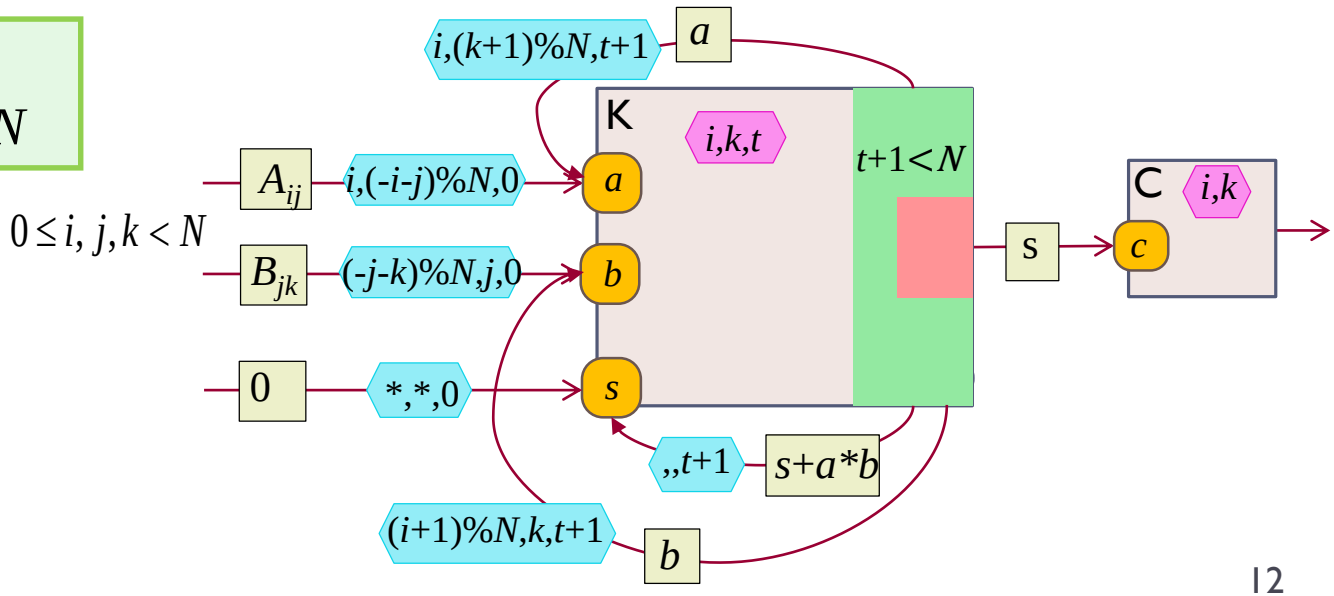
(b_{jk})#N ⟶ V.a⟨*,j,k⟩

Функции распределения:

$Place = (i,k)$

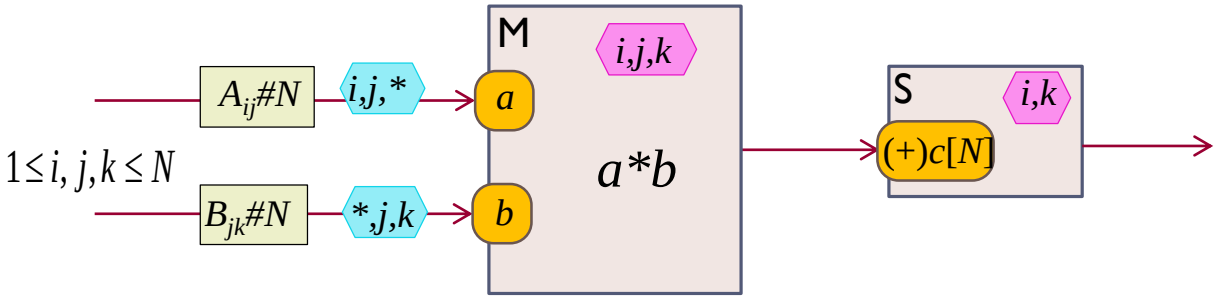
$Stage = t = (i+j+k)\%N$

Алгоритм Кэннона:



NxN Matrix Multiplication

Основное определение алгоритма:



Текстовая форма:

```
vconst N;
node A(float a,float b)⟨i,j,k⟩ {
    a*b ⟶ S.c⟨i,k⟩;
}
node S(float reduce(+)c[N])⟨i,k⟩{
    c ⟶ …
}
```

Засылка данных:

(a_{ij})#N ⟶ V.a⟨i,j,*⟩

(b_{jk})#N ⟶ V.a⟨*,j,k⟩

Функции распределения:

Place = (*i*/*m*,*k*/*m*,*j*/*m*) для узла M и входа S.c

(*i*/*m*,*k*/*m*,0) для узла S

Stage = (*i*%*m*, *k*%*m*, *j*%*m*)

Блочный распределенный вариант.

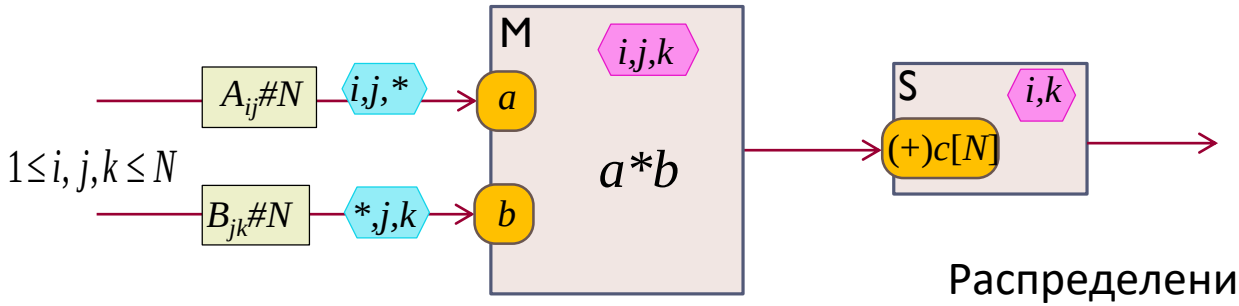
m – размер 3D блока, всего $\lceil N/m \rceil^3$ процессоров,

в каждом процессоре 1 блок,

внутри процессора обычный тройной цикл

NxN Matrix Multiplication

Основное определение алгоритма:



Распределение «по строкам» матрицы A

Функции распределения:

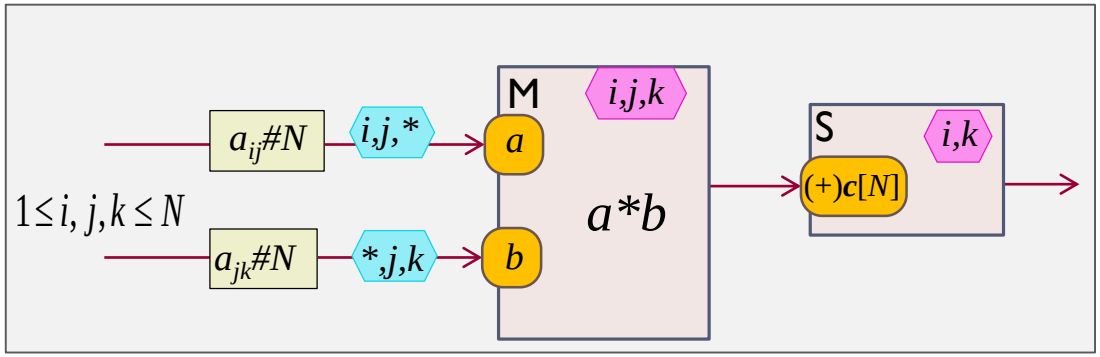
Place = *i/rows*
Stage = (*j/cols*, || , *i%rows*, *j%cols*, *k*)

mSize – размер блока строк (столбцов), в каждом процессоре 1 блок

```
/* Перебор групп столбцов */
  for (m = 0; m < MPI_CMCount; m++) {
    if (DFE_CMIndex == m)
      for (j = 0; j < mycols; j++)
        for (k = 0; k < mSize; k++)
          buf[k+j*mSize] = b[k][j];
 /* Пересылка m-ой группы столбцов матрицы b */
    if (m == MPI_CMCount-1) l = last_cols * mSize;
    else l = cols * mSize;
   MPI_Bcast(buf, l, MPI_DOUBLE, m, MPI_COMM_WORLD);
    if (m == MPI_CMCount-1) jmax = last_cols;
    else jmax = cols;
/* Вычисление своего блока матрицы */
    for (i = 0; i < myrows; i++)
      for (j = 0; j < jmax; j++)
        for (k = 0; k < mSize; k++)
          c[i][m*cols+j] += a[i][k] * buf[k+j*mSize];
  }
```

Умножение матриц NxN

Графическая форма алгоритма:



Текстовая форма:

```
vconst N;
node A(float a,float b)⟨i,j,k⟩ {
      a*b ⟶ S.c⟨i,k⟩;
}
node S(
  float reduce(+)c[N]) ⟨i,k⟩ {
  c ⟶ …
}
```

Засылка данных:

(a_{ij}) #N ⟶ V.a⟨i,j,*⟩

(b_{jk}) #N ⟶ V.a⟨*,j,k⟩

Варианты отображений (функции распределения по пространству и времени):

Place = (0)
Stage = (*i*,*k*,*j*)

Последовательный алгоритм

Place = (*i*/*m*,*k*/*m*,*j*/*m*)
Stage = (*i*%*m*,*k*%*m*,*j*%*m*)

Блочный распределенный алгоритм, *m* – размер блока

Place = (*i*,*k*)
Stage = *t* = (*i*+*j*+*k*)%*N*

Алгоритм Кэннона

Выводы

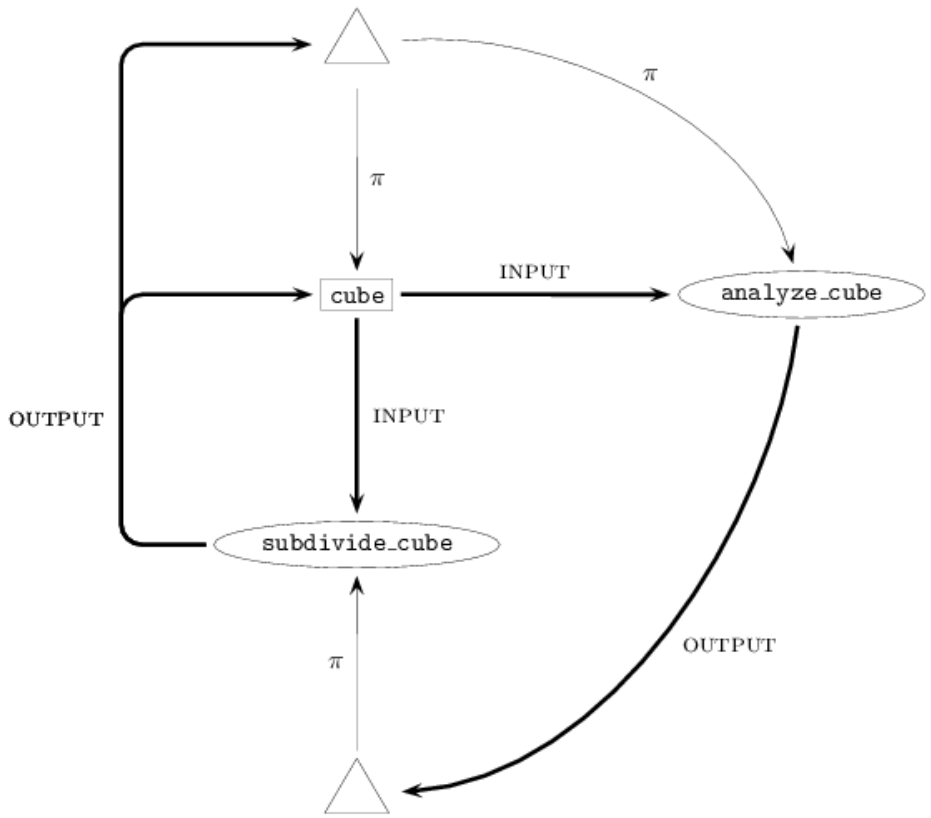
- Эффективный параллельный код = математический алгоритм + распределение по пространству/времени
- Как задавать распределения?
- В языке с индексами (тегами) – как функции от индексов.
- Можно начать с полиэдральных моделей как исходного языка программирования в парадигме раздачи, но расширить класс допустимых программ. Это и будет наш язык UPL
- Умножение матриц, задача N тел – аффинные программы.
- В метаязыке UPL выразим более широкий класс алгоритмов.
- Надо «научиться» порождать коды для разных платформ так же, как это сейчас делают для полиэдральных моделей аффинных программ.

Проблемы и перспективы.

- Чтобы обеспечить универсальнось, платформо-независимость и эффективность необходимо
  - Иметь развитую систему эквивалентных преобразований, способность их применять, проверять эквивалентность
  - Уметь оценивать сложность (эффективность) вариантов отображений вычислительных схем.
  - Уметь выбирать оптимальное решение для данной платформы и данного приложения.

- Спасибо. Вопросы?

Элементы Concurrent Collections (CnC)



Три типа коллекций:
- Тэги (controls)
- Данные (items)
- Вычисления (steps)

Steps:

- Запрашивают items
- Создают items и controls

Controls:

- Создают steps

Items:

- Уведомляют запросившие steps о готовности

Это схема алгоритма N-body Barness-Hut
Иллюстрация заимствована из отчета

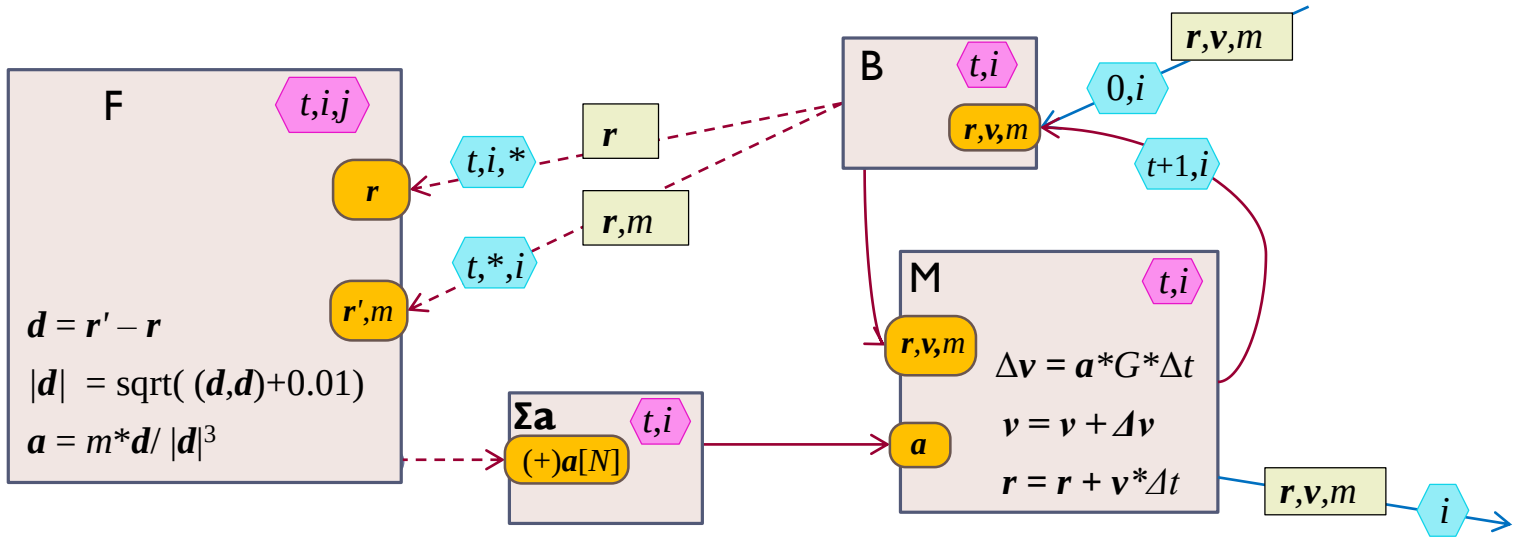
Kathleen Knobe, Carl D. Offner ,Tstreams: A model of parallel computation (preliminary report). Technical Report HPL, 25 July, 2005, HP Labs

на котором основана модель CnC

Отличия UPL и CnC

	CnC	UPL
Типы коллекций	Steps, Items, Controls	Узлы = Steps
Парадигма	Сбора	Раздачи
Активация узлов (steps)	По специальной операции	По приходу всех items
Активности движутся	По дереву	По ациклическому графу
Узел (Step)	Выдает items и controls Запрашивает items	Посылает item на входы узлов
Данные (items)	Пассивны (надо запрашивать)	Активны (приходят сами на вход и активируют узел)

All-pairs N-body problem



Обозначения:

(+)$a[N]$ – суммирующий вход, N – число слагаемых

i,j – номера тел

$\boldsymbol{r}$, $\boldsymbol{r}'$ – координаты тел (вектор),

$\boldsymbol{d}$ – дистанция между телами (вектор)

$\boldsymbol{v}$ – скорость (вектор)

$\Delta\boldsymbol{v}$ – приращение скорости

t – номер шага: $0,1,\ldots,t_{max}$,

Δt – шаг по времени (константа)

N – количество тел (константа)

G – гравитационная постоянная (константа)

Узлы:

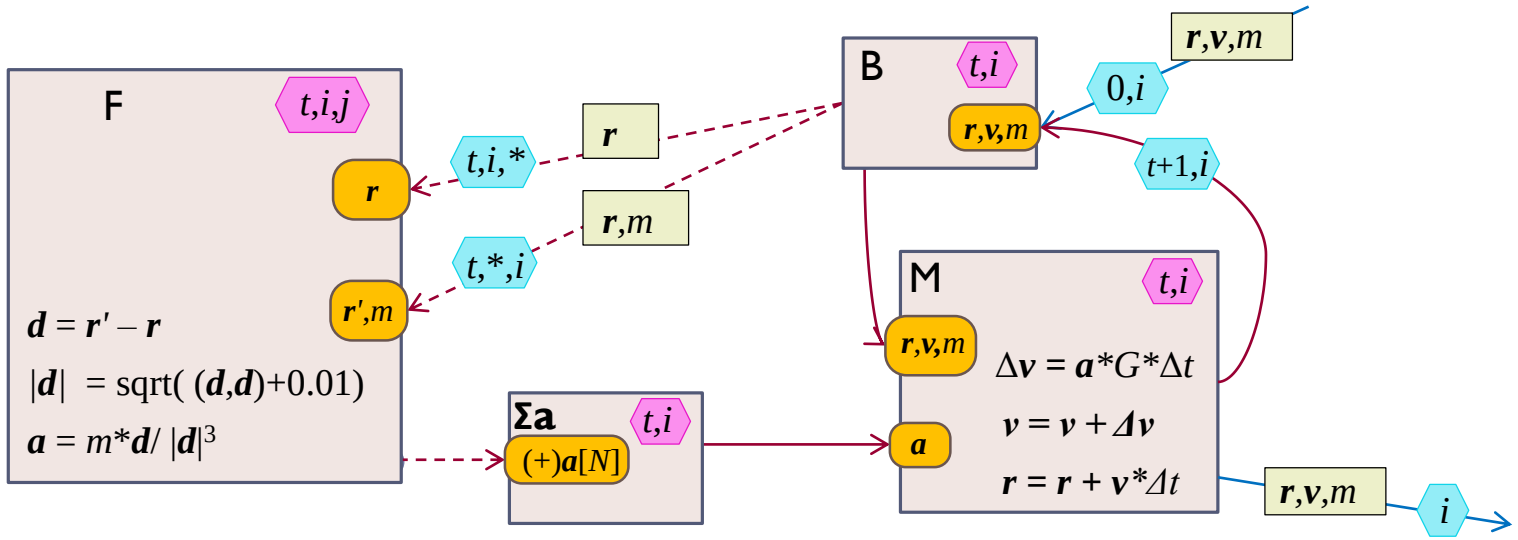
F – вычисляет поле в точке $\boldsymbol{r}$ от другого тела в точке $\boldsymbol{r}'$

Σa – вычисляет суммарное поле $\boldsymbol{a}$ в позиции тела i

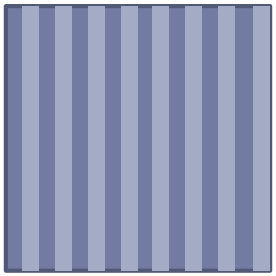
M – вычисляет новые координаты и скорости ($\boldsymbol{r}$,$\boldsymbol{v}$) тела i

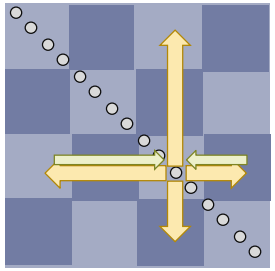
B – передает данные о теле i на узлы **F** и **M**

All-pairs N-body problem. Распределение вычислений

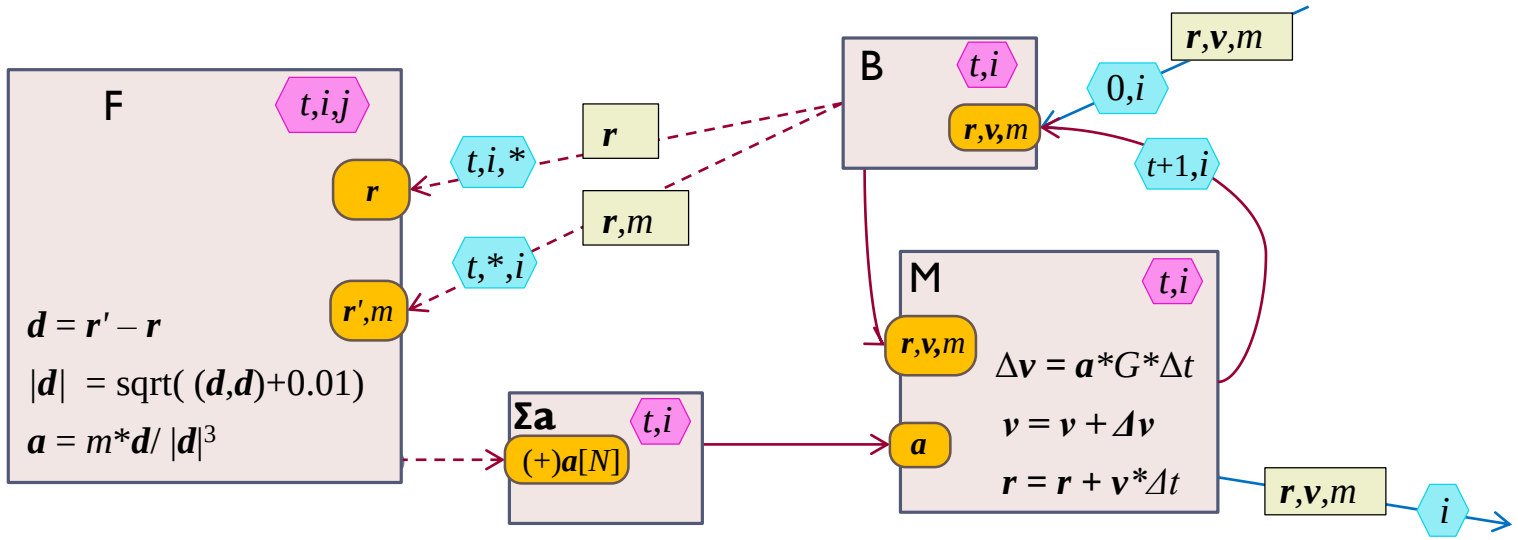


- Варианты функций распределения:
  а) $place = i/m$ б) $place = [\ i/m\ ,\ j/m\ ]$, $m=\lceil N/\sqrt{P} \rceil$ для узла F,
  $stage = t$ $[\ i/m\ ,\ i/m\ ]$ для остальных узлов
- Объем коммуникаций на одну итерацию:
  а) $O(N\,P)$ б) $O(N\sqrt{P})$

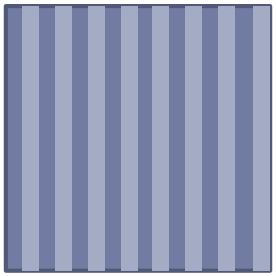


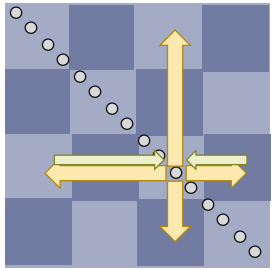


All-pairs N-body problem. Распределение вычислений



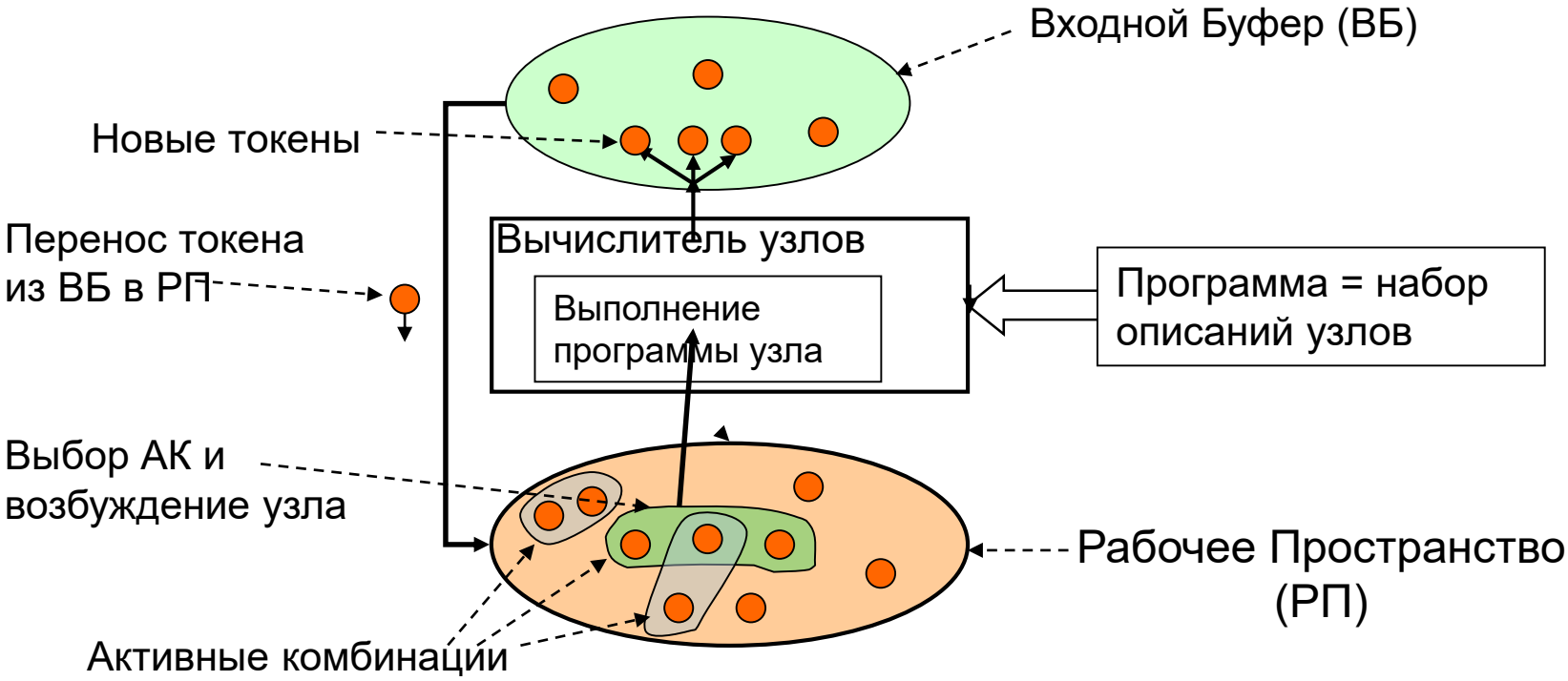
- Варианты функций распределения:
  а) $place = i/m$ б) $place = [\ i/m\ ,\ j/m\ ]$, $m=\lceil N/\sqrt{P} \rceil$ для узла F,
  $stage = t$ $[\ i/m\ ,\ i/m\ ]$ для остальных узлов
- Объем коммуникаций на одну итерацию:
  а) $O(N\,P)$ б) $O(N\sqrt{P})$





Важное условие: каждая пара токенов встречается **ровно** в одном ядре

Абстрактный вычислитель



- В Рабочем Пространстве (РП) обнаруживаются активные комбинации. Некоторые из них возбуждают соответствующие им узлы. Использованные токены обычно удаляются из РП.
- Вычислитель выполняет программы возбужденных узлов, в результате чего создаются новые токены. Последние вносятся во Входной Буфер (ВБ), откуда по одному в некотором неопределенном порядке поступают в РП.

Проблема детерминизма

- В нашей модели вычислений детерминизма нет (в отличие, скажем, от CnC, где допускаются только детерминированные программы и это проверяется статически).
- Большинство задач на практике – детерминированы
- Статическая проверка детерминизмы – интересная задача
- Есть простой динамический критерий детерминизма:
  - *Будем оставлять все стираемые токены с пометкой, и следить, не образуется ли активная комбинация с участием помеченного токена. Если этого ни разу не случилось, то данное вычисление детерминировано!*

Схема алгоритма SSSP. Наивный вариант (без синхронизации)

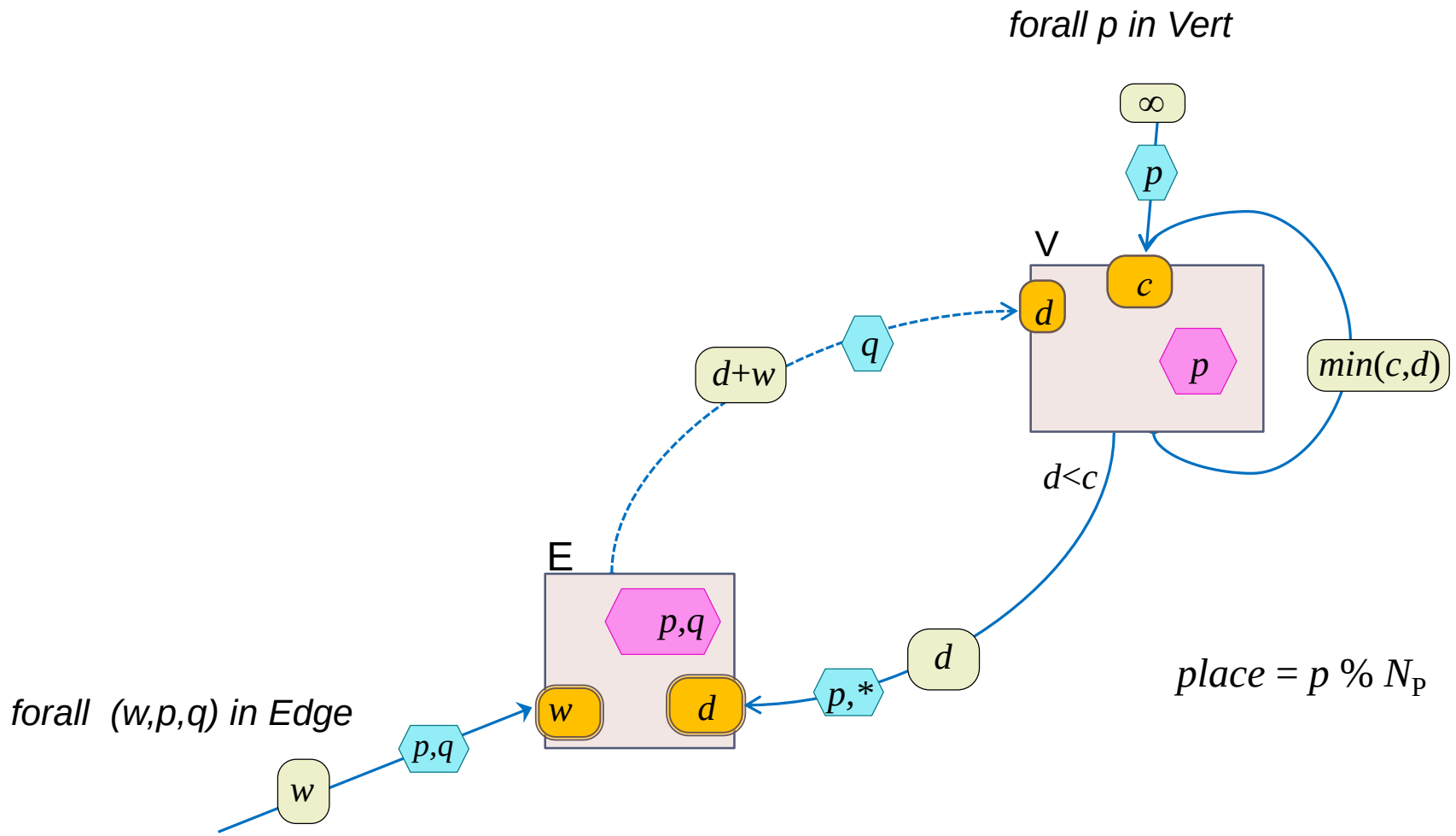
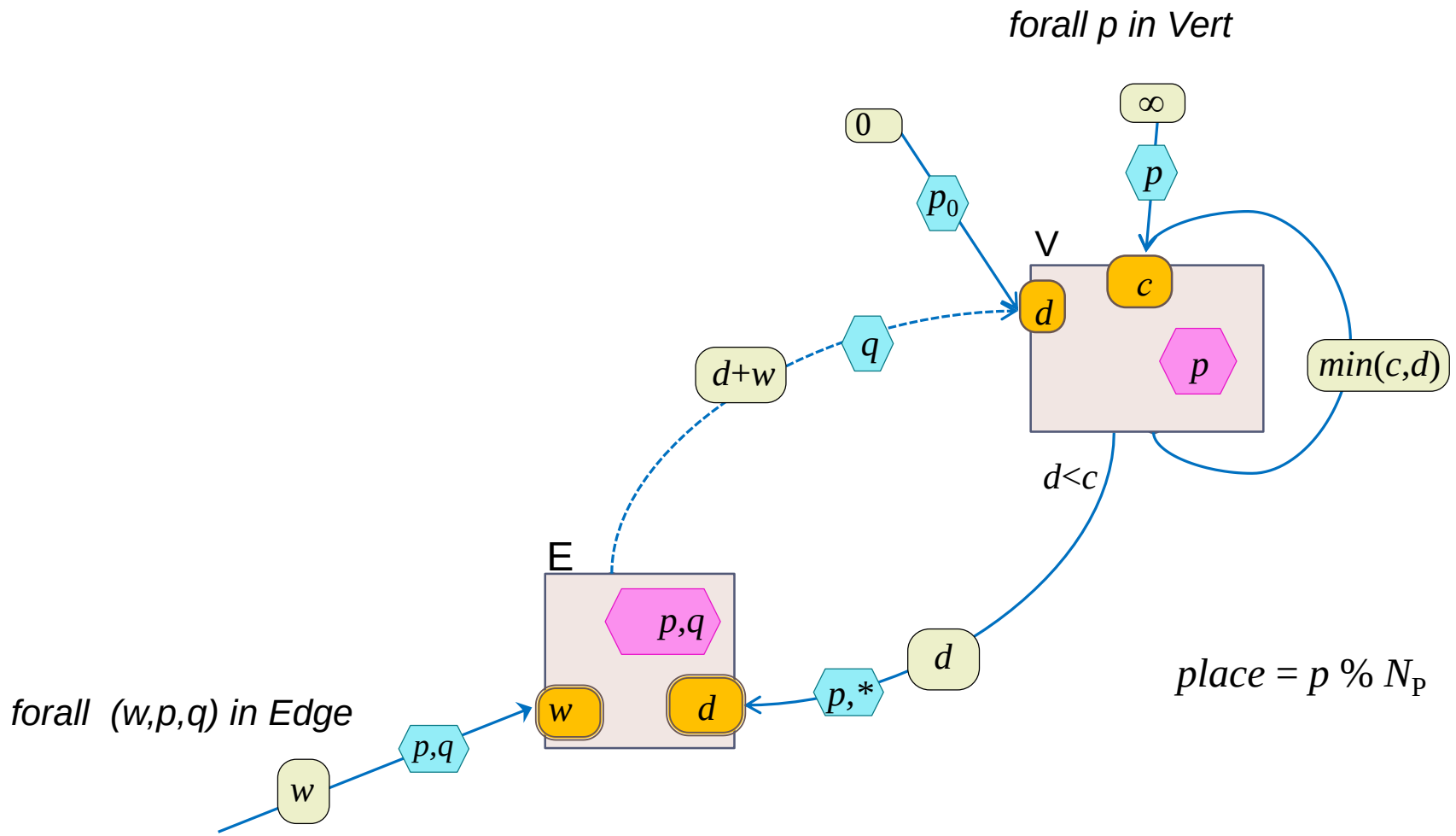


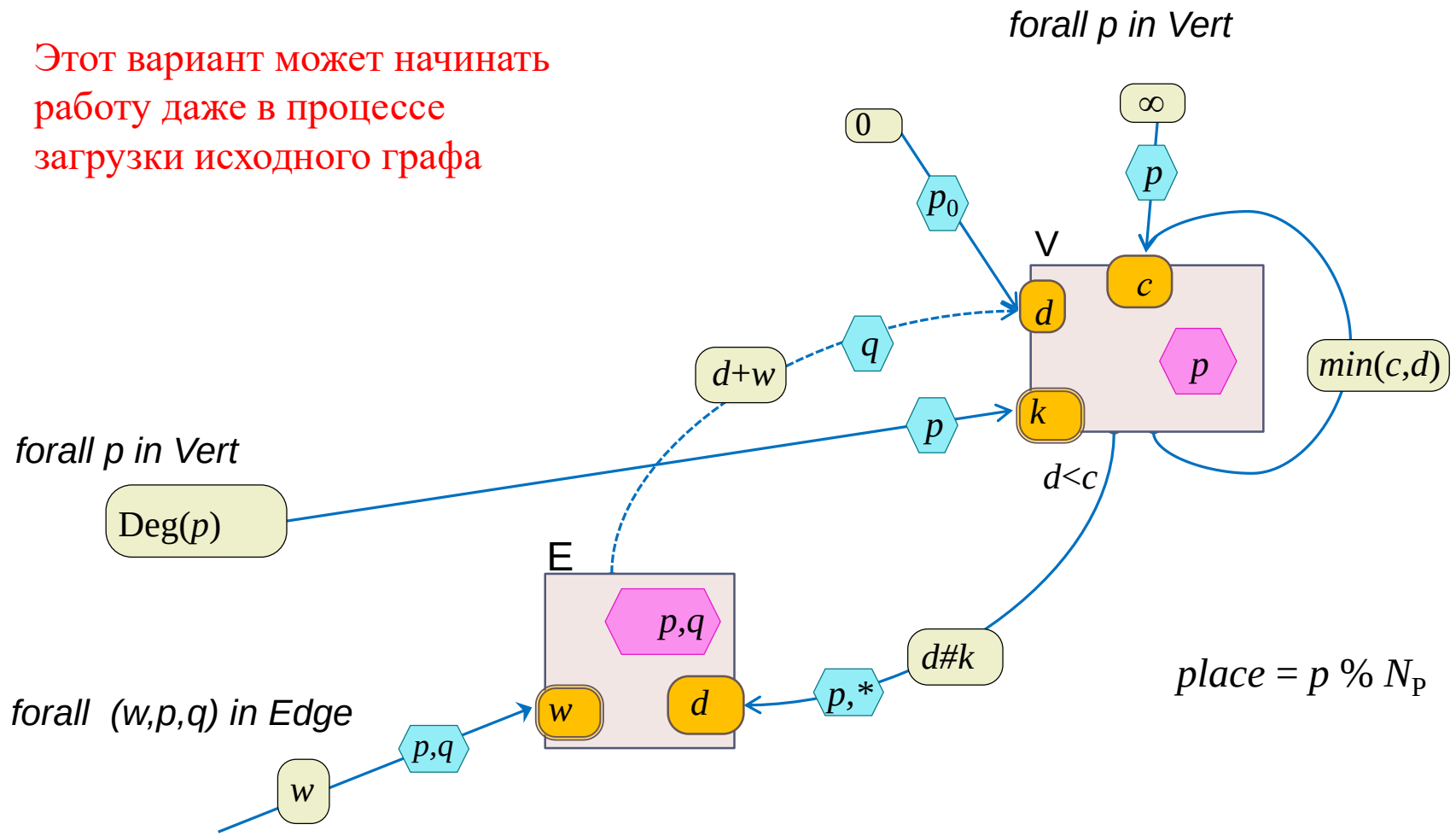
Схема алгоритма SSSP. Наивный вариант (без синхронизации)



По завершению всякой активности кратчайшие расстояния от p_0 до p, будут находиться в V.c<p> . Возможны многократные проходы по дугам.
Пунктиром обозначены передачи по дугам. Только они могут быть нелокальными.
Узлы распределяются по ядрам по полю p взятием остатка от деления на число ядер N_P.

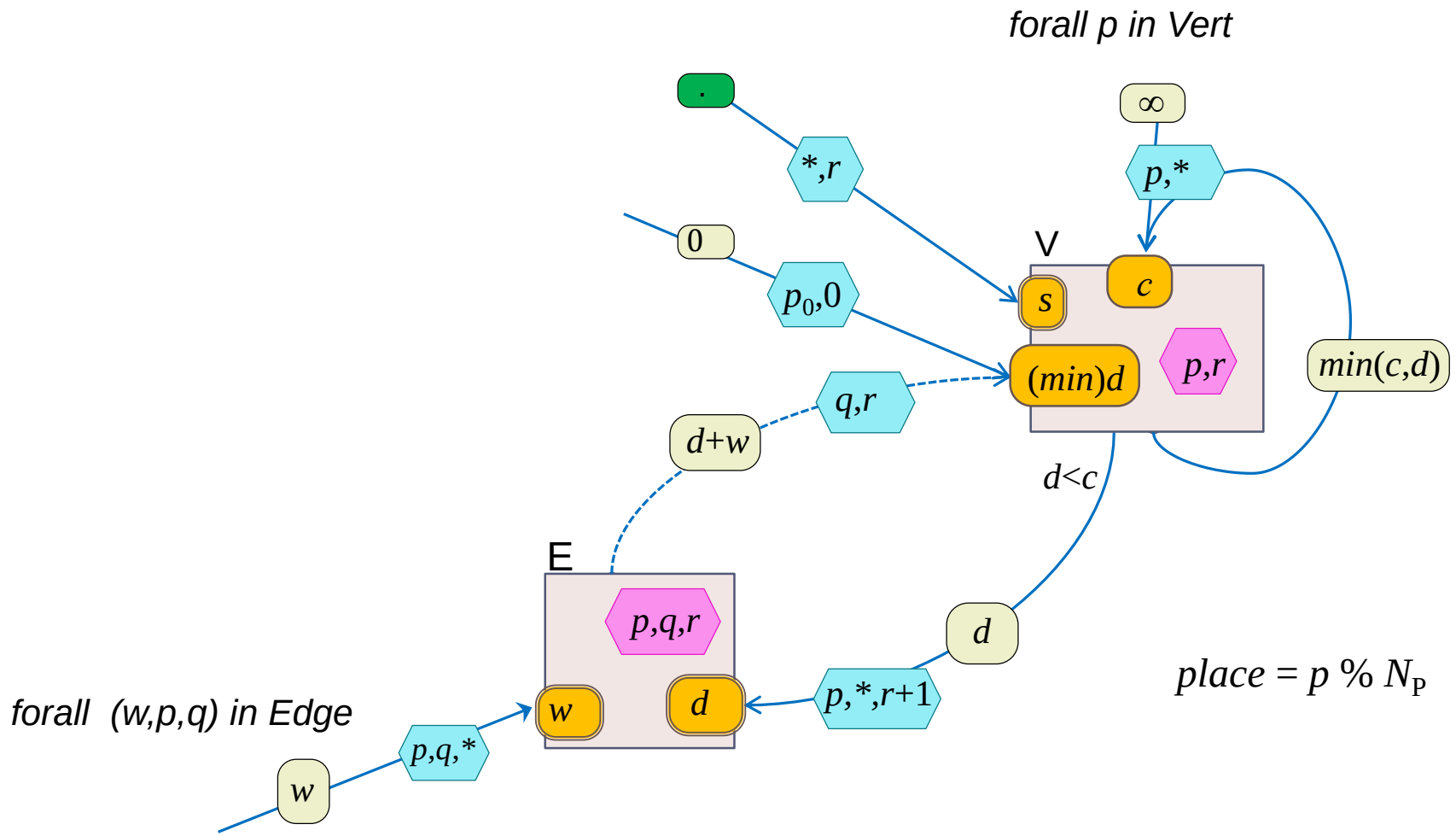
Схема алгоритма SSSP. Наивный вариант (без синхронизации)

Этот вариант может начинать работу даже в процессе загрузки исходного графа



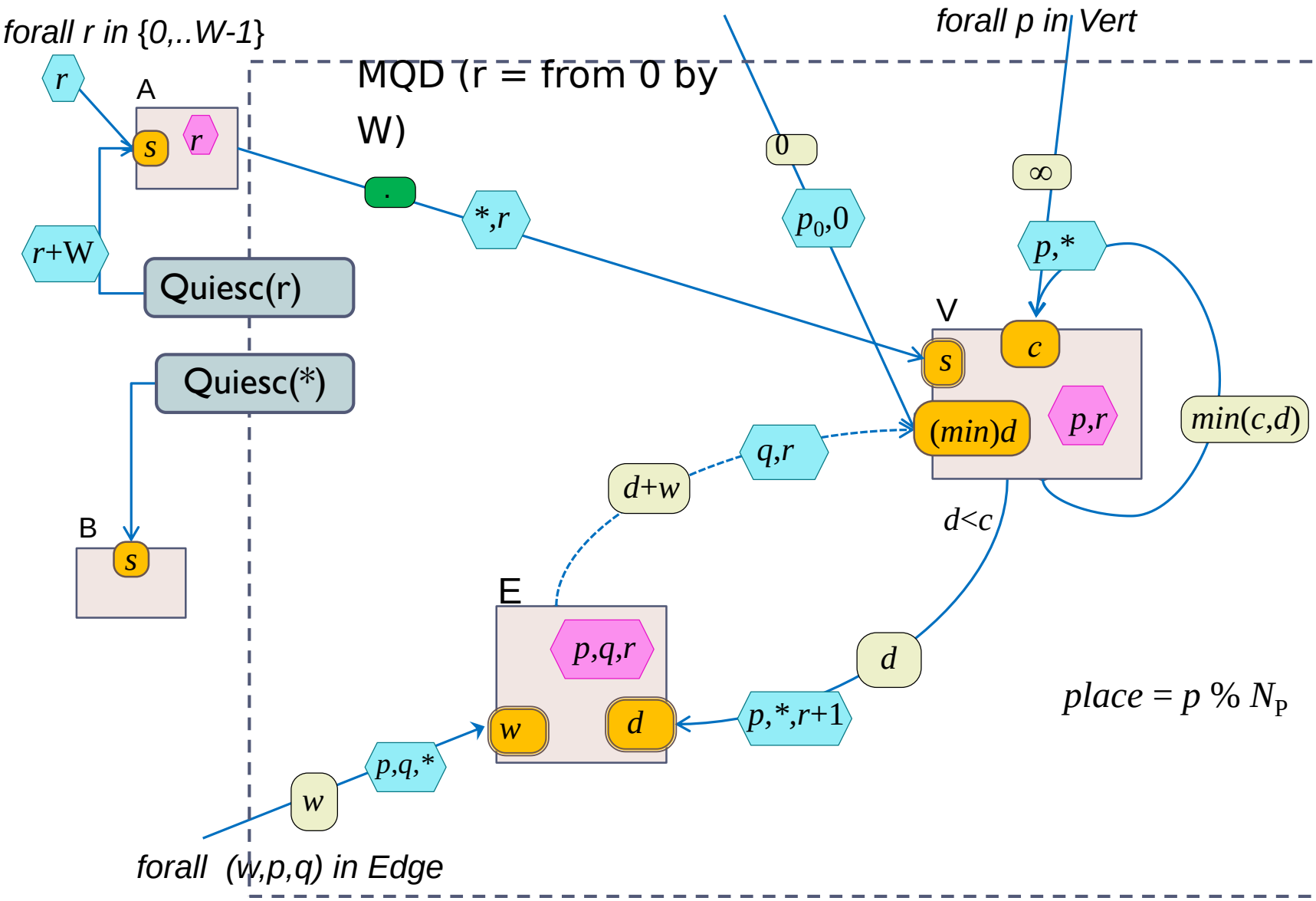
По завершению всякой активности кратчайшие расстояния от p_0 до p, будут находиться в V.c<p> . Возможны многократные проходы по дугам.
Пунктиром обозначены передачи по дугам. Только они могут быть нелокальными.
Узлы распределяются по ядрам по полю p взятием остатка от деления на число ядер N_P.

Схема SSSP с внешней синхронизацией по уровням BFS.

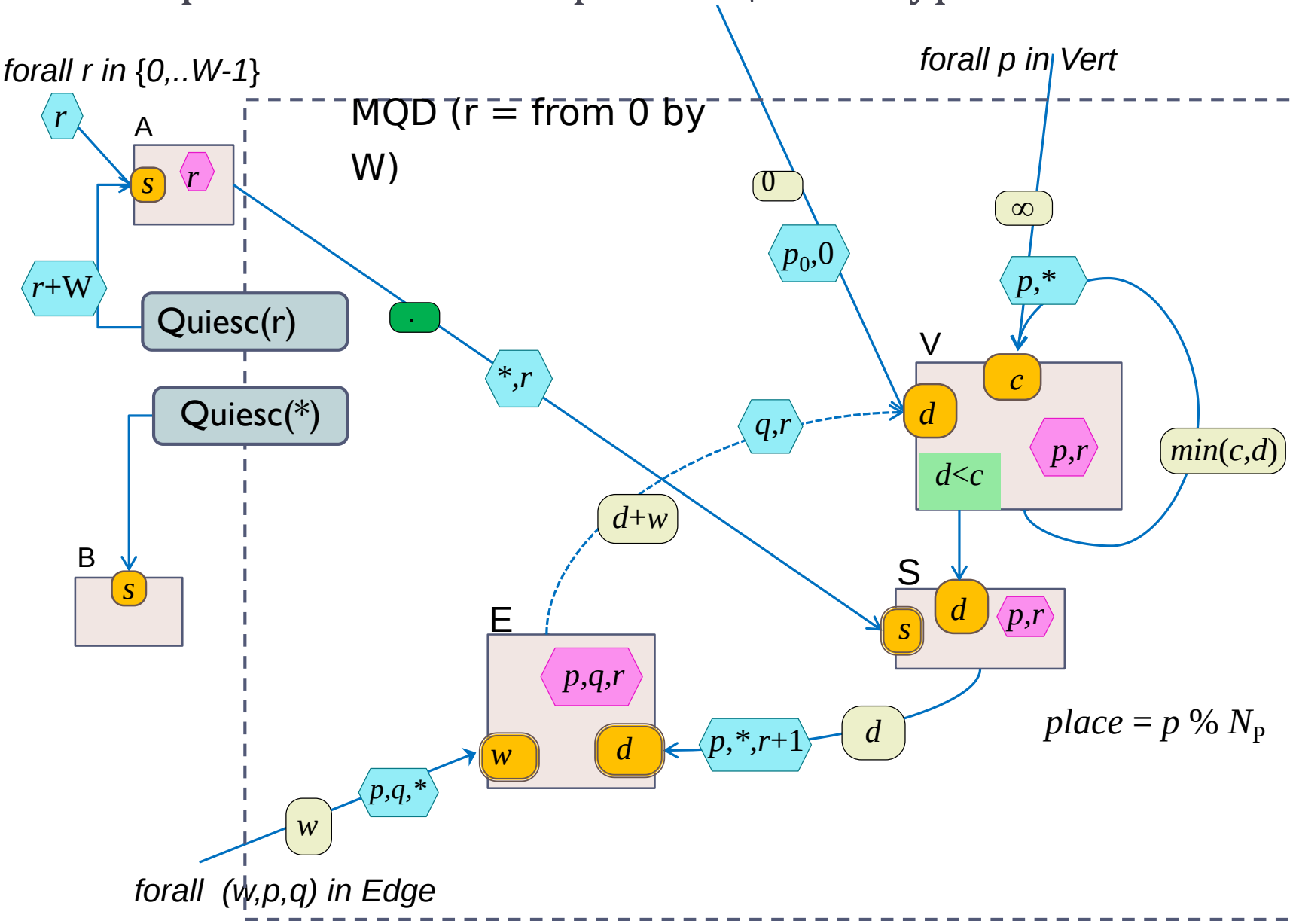


Окружение реализует *многоуровневый детектор тишины,* который порождает токен ()⟶V.s{*r*+w,*}, когда на уровне *r* фиксируется завершение активности (w – ширина окна активности).
В начале работы создаются w таких токенов для r=0,1…,w-1. Для стандартного варианта w=1.
На входе V.d выполняется редукция через *min.*

Алгоритм SSSP с синхронизацией по уровням BFS с MQD



Алгоритм SSSP с синхронизацией по уровням BFS с MQD



МРТ: многоуровневый распознаватель тишины (MQD: multilevel quiescence detector)

Глобальная синхронизация осуществляется посредством механизма распознавания тишины (quiescence detection), который встроен в исполняющую систему. Метод реализации – распределенные счетчики.

Простой распознаватель тишины основан на локальном подсчете токенов, посланных и принятых во всех ядрах, и глобальном суммировании этих счетчиков. Нулевая сумма является признаком тишины, то есть завершения всех активных действий.

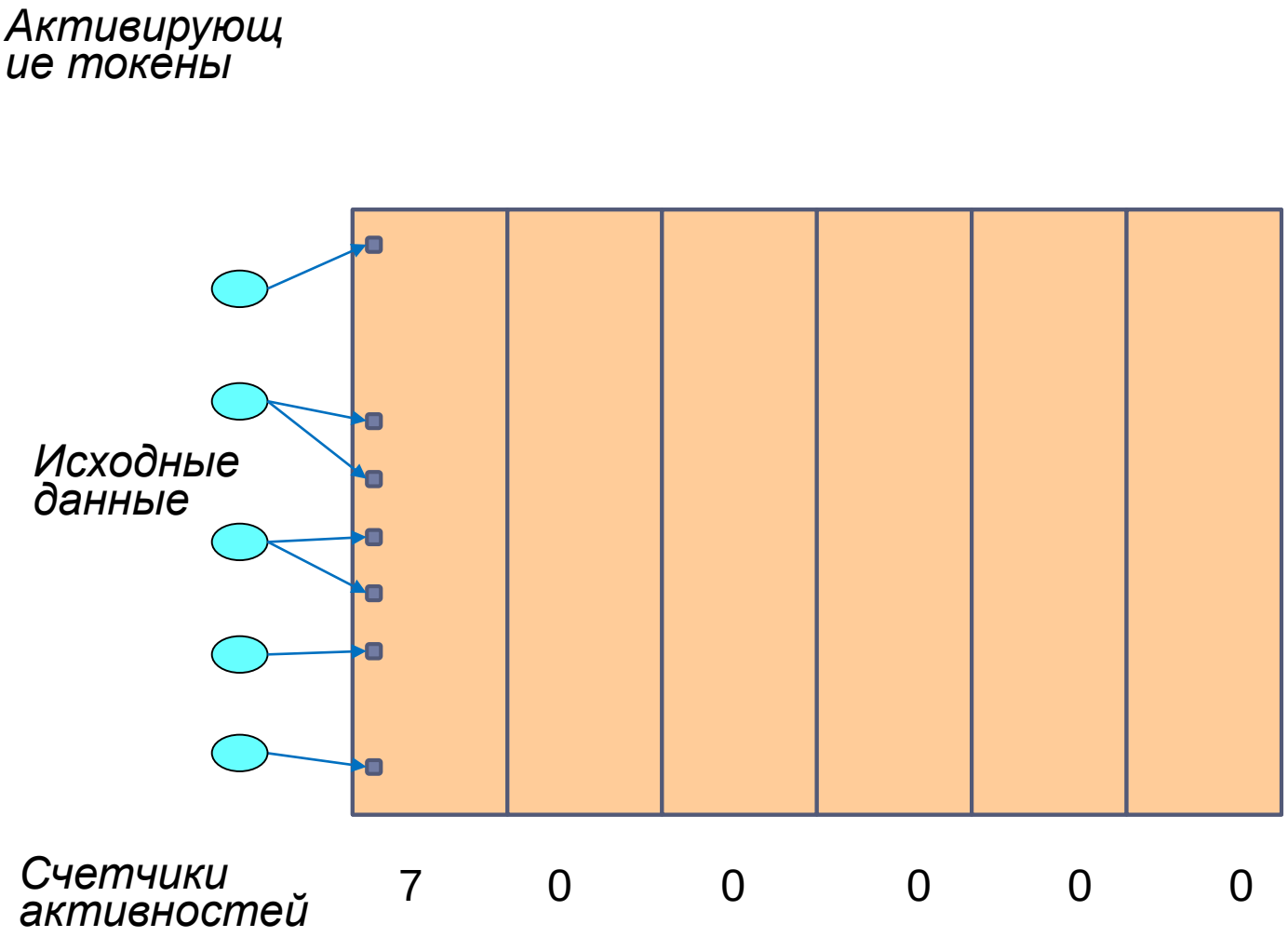
Наше решение задачи SSSD использует многоуровневый детектор тишины. В нем подсчет ведется раздельно по всем значениям выделенного поля r. (Предполагается, что активность для r_1 может породить только активность для $r_2 \geq r_1$).

Токены с r=* учитываются особо: их сумма виртуально прибавляется ко всем счетчикам, так что пока она не нуль, ни одна сумма для конкретного r не может быть расценена как нулевая.

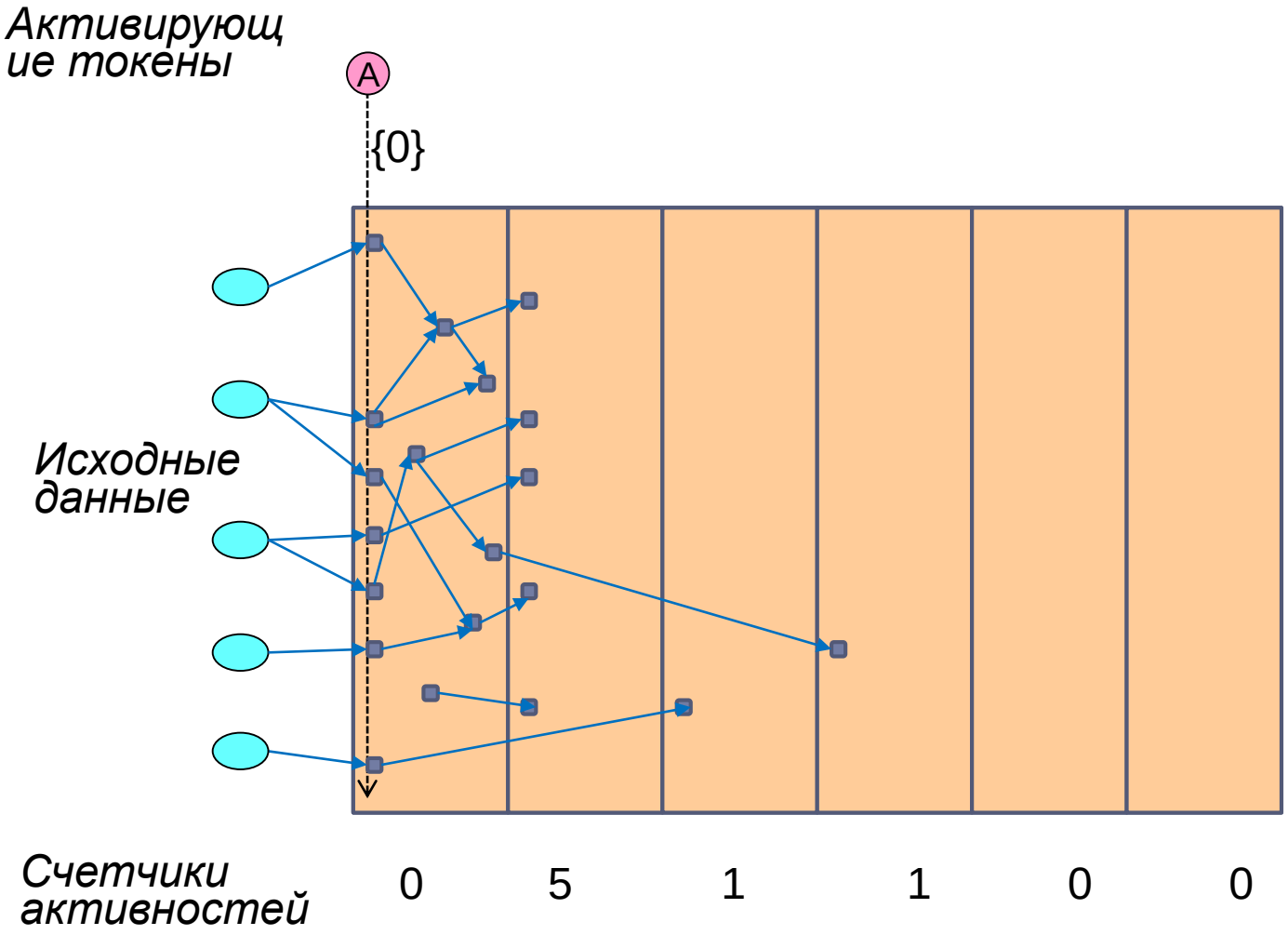
МРТ глобально определяет минимальное значение r_m, в котором имеются зародыши активности, и активизирует их заданным образом (у нас это посылка #∞→S.s{r_m,*}), или сообщает, что ненулевых r больше нет.

Возможна активизация сразу для W последовательных значений r, начиная с r_m: r_m, r_m+1,…, r_m+W-1 (избегая повторных и бесполезных активаций). SSSD допускает использование любого W>0.

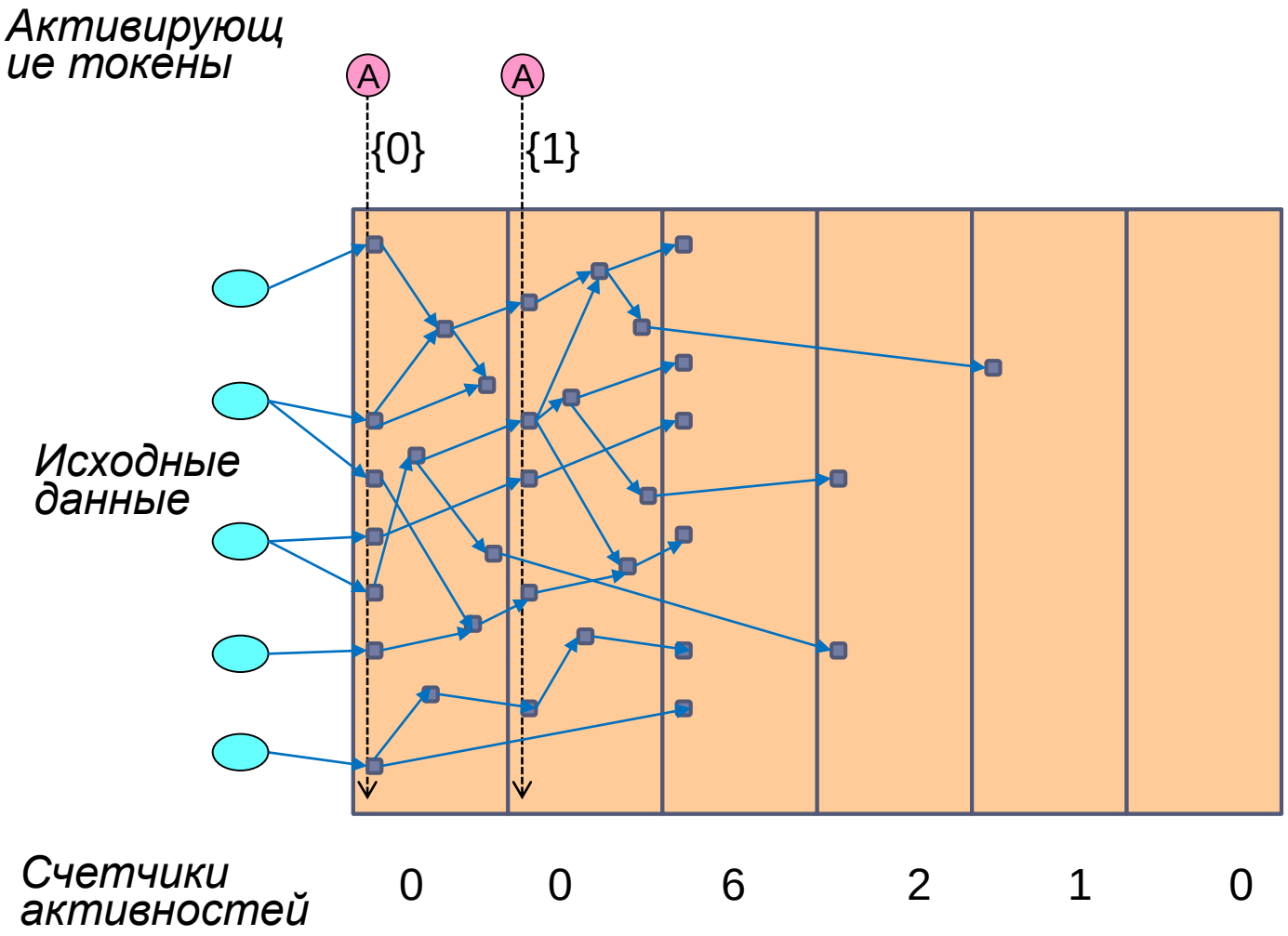
Ход вычисления под управлением MPT



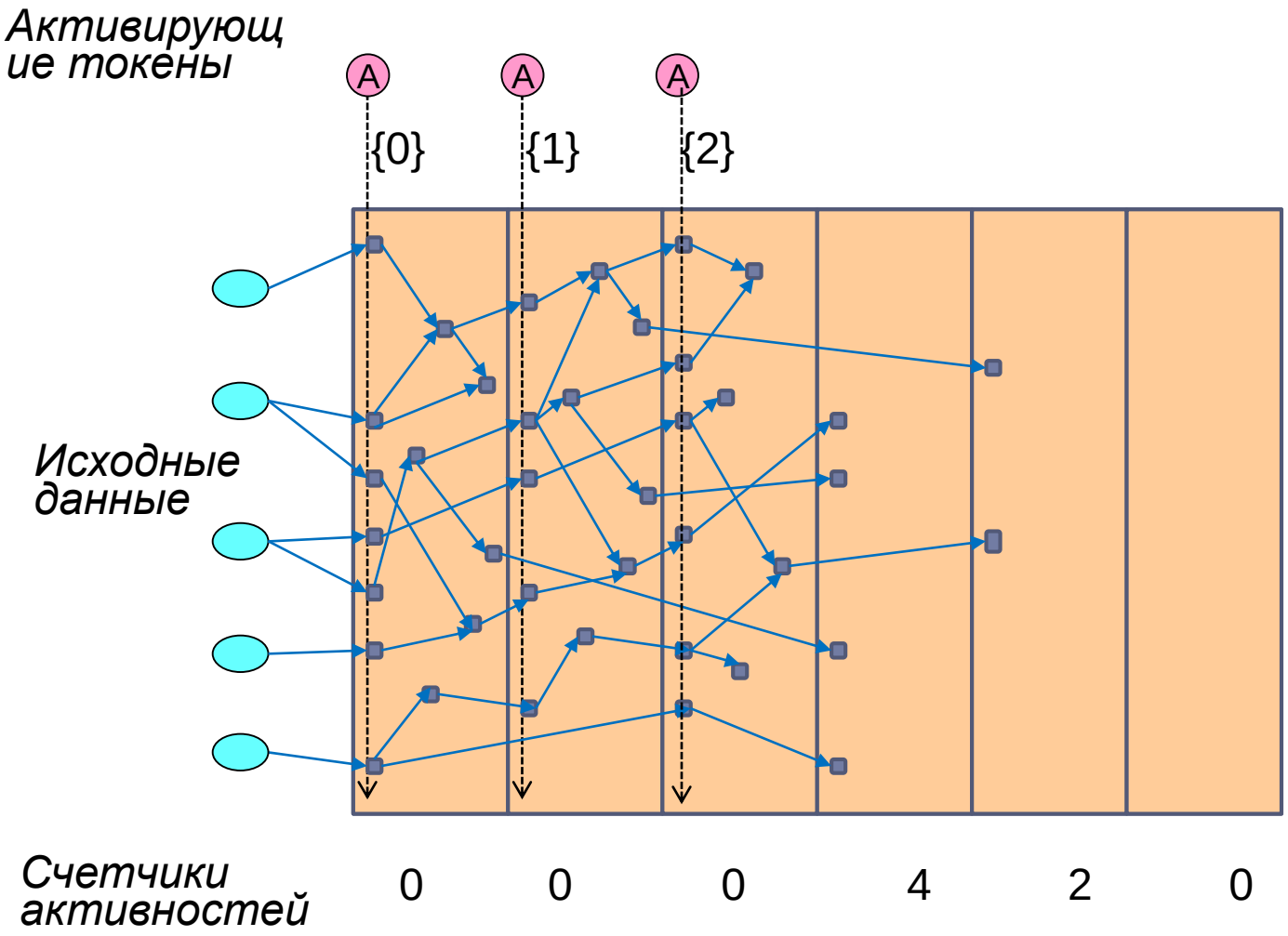
Ход вычисления под управлением МРТ



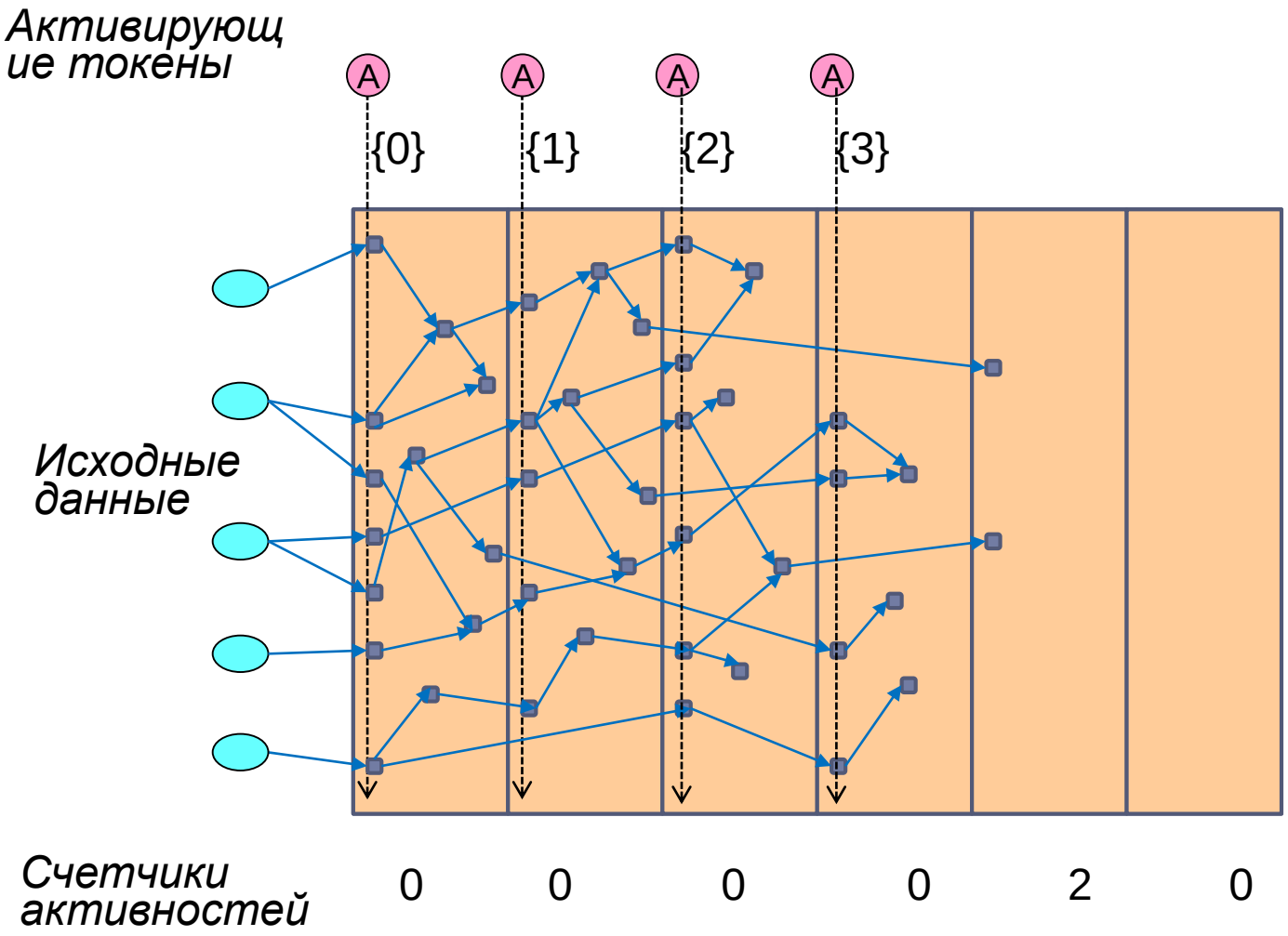
Ход вычисления под управлением MPT



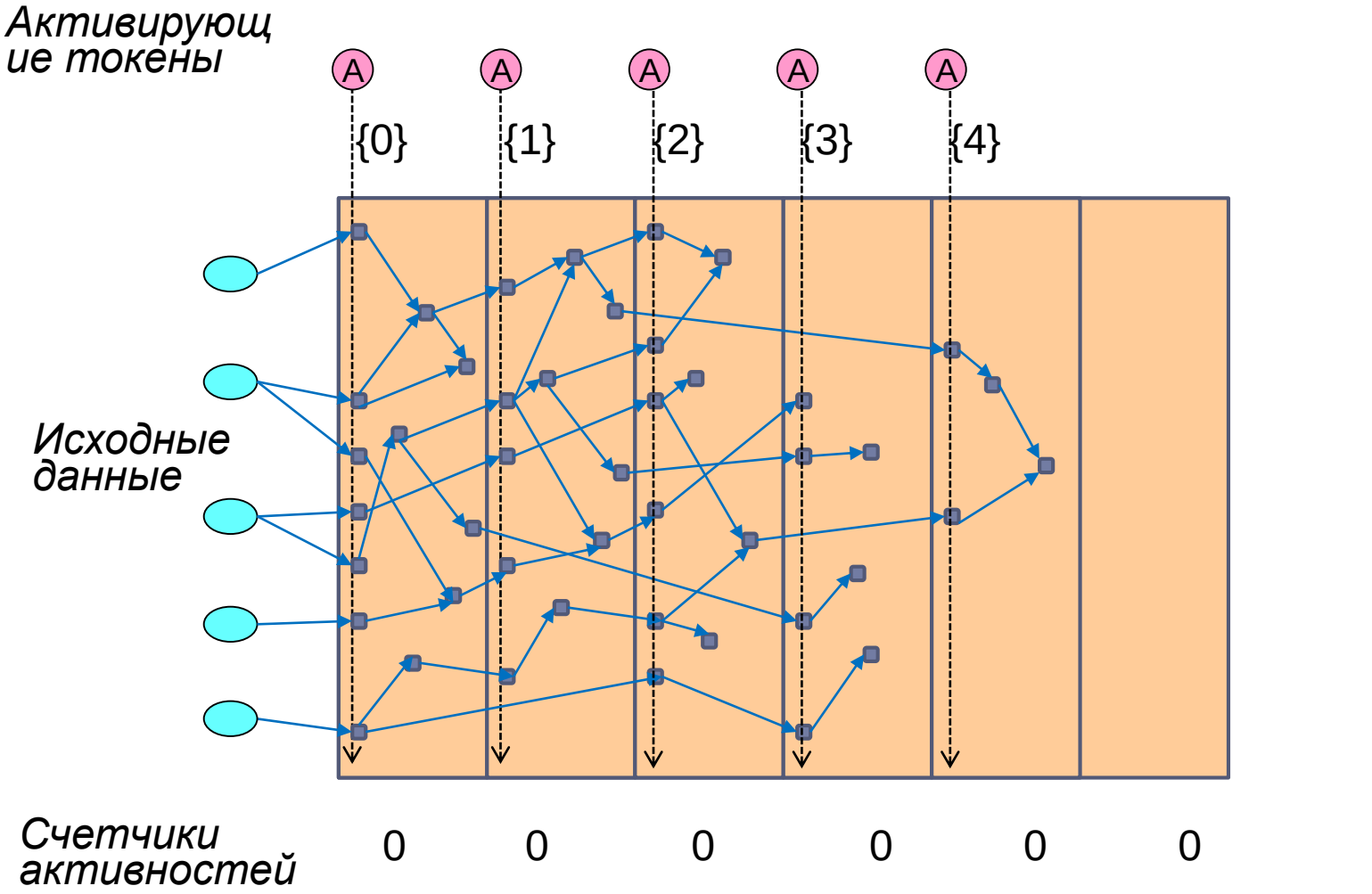
Ход вычисления под управлением MPT



Ход вычисления под управлением MPT

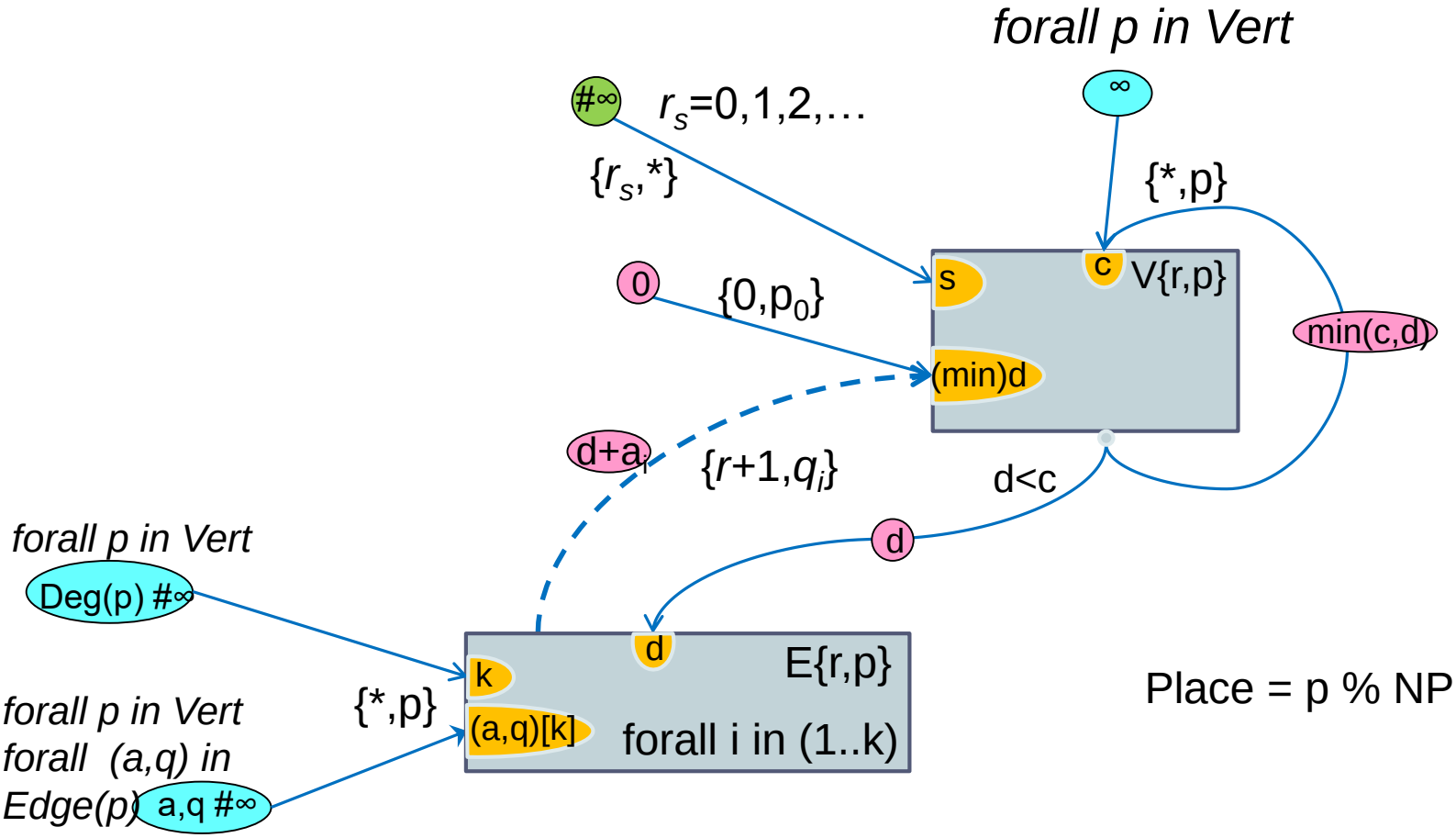


Ход вычисления под управлением МРТ



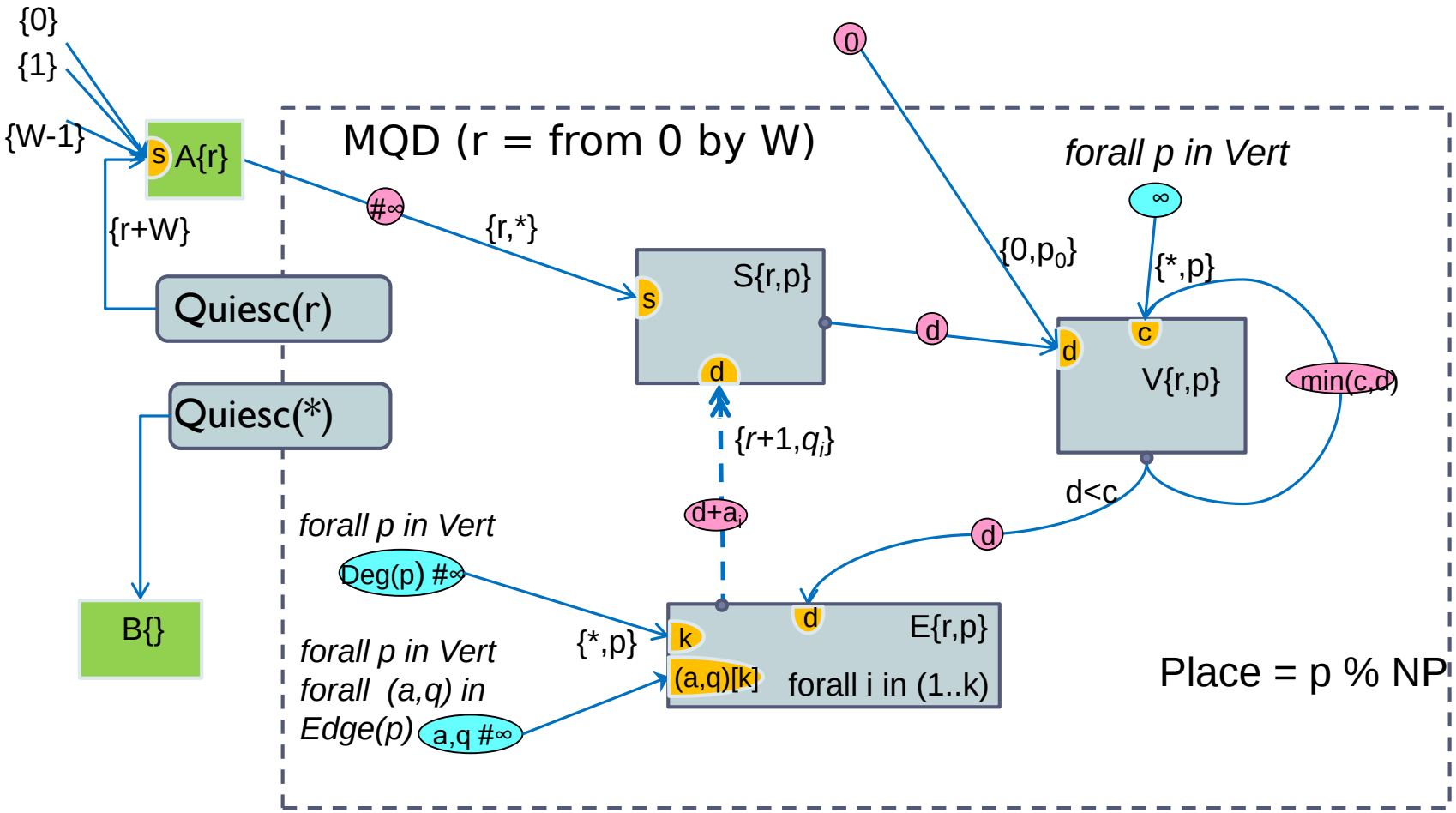
КОНЕЦ!

Схема алгоритма.
Базовый вариант с синхронизацией по уровням BFS.



Окружение реализует *многоуровневый детектор тишины,* который порождает токен ()⟶V.s{*r*+w,*}, когда на уровне *r* фиксируется завершение активности (w – ширина окна активности).
В начале работы создаются w таких токенов для r=0,1…,w-1. Для стандартного варианта w=1.
На входе V.d выполняется редукция через *min.*

Схема алгоритма. Синхронизация по уровням BFS. Вариант с MPT.



Многоуровневый детектор тишины активирует узел A{*r*}, когда на уровне *r* фиксируется завершение активности (w – ширина окна активности). Для стандартного варианта w=1.
В начале работы активируются первые w таких узлов для r=0,1…,w-1.
По полной тишине (во всех уровнях) активируется узел B{}.

Программа на PolyDFL

Исходные данные и инициализация:

k → E.k{*,*p*}	– для всех вершин p, из которых выходит k дуг
(*a*,*q*) → E.e{*,*p*};	– для всех дуг веса *a* из вершины *p* в вершину *q*
MAXREAL → V.c{*,*p*};	– для всех вершин *p* (признак недостигнутости)
0 → V.d{p_0};	– для начальной вершины p_0

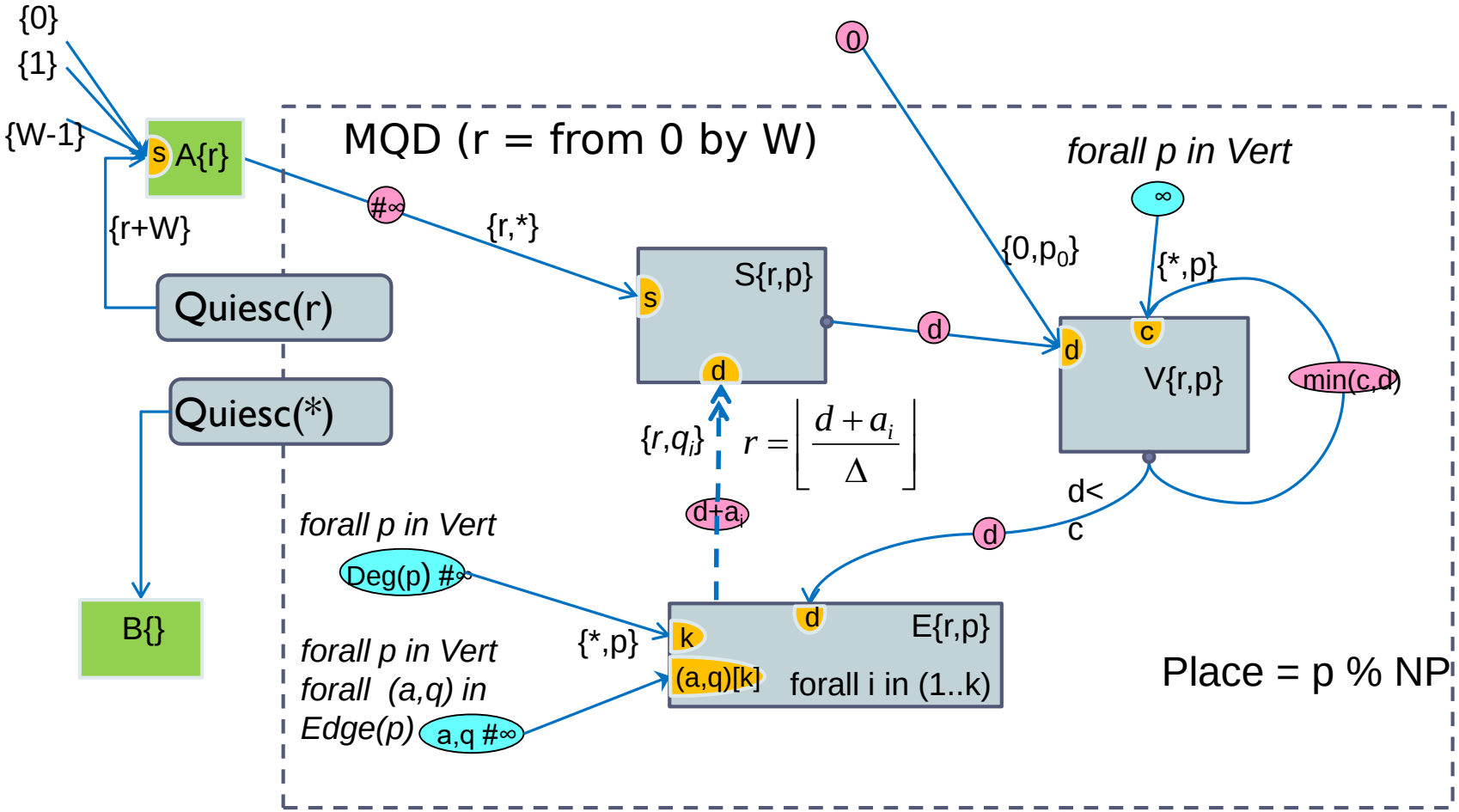
Описания узлов на PolyDFL:

```
varconst  Δ:real; W:int;
MQD SSSD (r; W; () → V.s ⟨r,*⟩; () → B⟨⟩);
      node S(sav void s, float (min) d) ⟨r,p⟩;
         d → V.d⟨r,p⟩;
      node V(float d, float c) ⟨r,p⟩;
        min(d,c) → V.c⟨*,p⟩;
        if (d<c)
            d → E.d⟨r,p⟩;
      node E(sav e(float a, int q)[k], sav int k, float d) ⟨r,p⟩;
        for i :=1 to k do
          let v := d+a[i] in
            v → S.d ⟨ r+1, q[i] ⟩;
End MQD
```

Активация процесса под контролем многоуровневой тишины (MQD) внутри некоторого узла:

```
startMQD SSSD
   for p in G.Nodes
      p.degree → E.k⟨*,p⟩;
      for e in p.Nbrs
          (e.wei,e.nei) → E.e⟨*,p⟩;
      +Inf → V.c⟨*,p⟩;
   0.0 → V.d⟨0,p0⟩;
```

Схема алгоритма. Вариант с МРТ с синхронизацией по Δ-кольцам.



Многоуровневый детектор тишины активирует узел A{*r*}, когда на уровне *r* фиксируется завершение активности (w – ширина окна активности). Для стандартного варианта w=1.
В начале работы активируются первые w таких узлов для r=0,1…,w-1.
По полной тишине (во всех уровнях) активируется узел B{}.

Программа на PolyDFL

Исходные данные и инициализация:

k → E.k{*,*p*}	– для всех вершин p, из которых выходит k дуг
(*a*,*q*) → E.e{*,*p*};	– для всех дуг веса *a* из вершины *p* в вершину *q*
MAXREAL → V.c{*,*p*};	– для всех вершин *p* (признак недостигнутости)
0 → V.d{p_0};	– для начальной вершины p_0

Описания узлов на PolyDFL:

```
varconst  Δ:real; W:int;
MQD SSSD (r; W; () → V.s ⟨r,*⟩; () → B⟨⟩);
      node S(sav void s, float (min) d) ⟨r,p⟩;
         d → V.d⟨r,p⟩;
      node V(float d, float c) ⟨r,p⟩;
        min(d,c) → V.c⟨*,p⟩;
        if (d<c)
            d → E.d⟨r,p⟩;
      node E(sav e(float a, int q)[k], sav int k, float d) ⟨r,p⟩;
        for i :=1 to k do
          let v := d+a[i] in
            v → S.d ⟨ trunc(v/Δ), q[i] ⟩;
End MQD
```

Внутри некоторого узла:

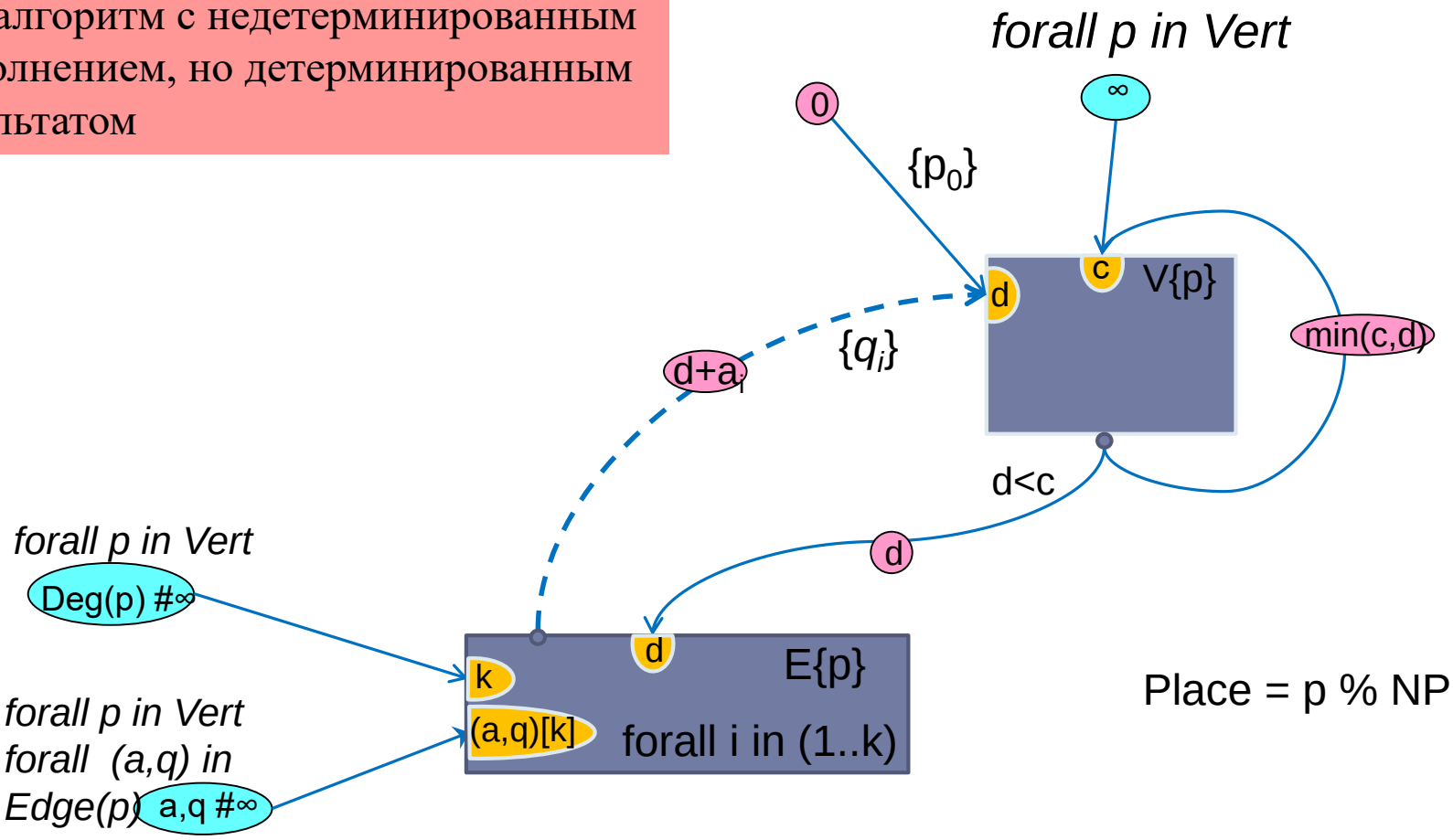
```
startMQD SSSD
  for p in G.Nodes
     p.degree → E.k⟨*,p⟩;
     for e in p.Nbrs
        (e.wei,e.nei) → E.e⟨*,p⟩;
     +Inf → V.c⟨*,p⟩;
  0.0 → V.d⟨0,p0⟩;
```

Резюме по МРТ

- Для глобальной синхронизации имеется мощный аппарат распознавания тишины, избавляющий программиста от проблем.
- Аппарат распознавания многоуровневой тишины с окном ширины W удобен для организации глобального упорядочения.
- Варьируя размер шага h и ширину окна W, можно находить оптимальное решение, балансируя объем лишней работы и параллелизм.
- Нами реализован эмулятор, для которого программа кодируется в C. Пока в нем имеется только простой распознаватель тишины. Используя его в цикле, мы имитируем многоуровневый с окном W=1.

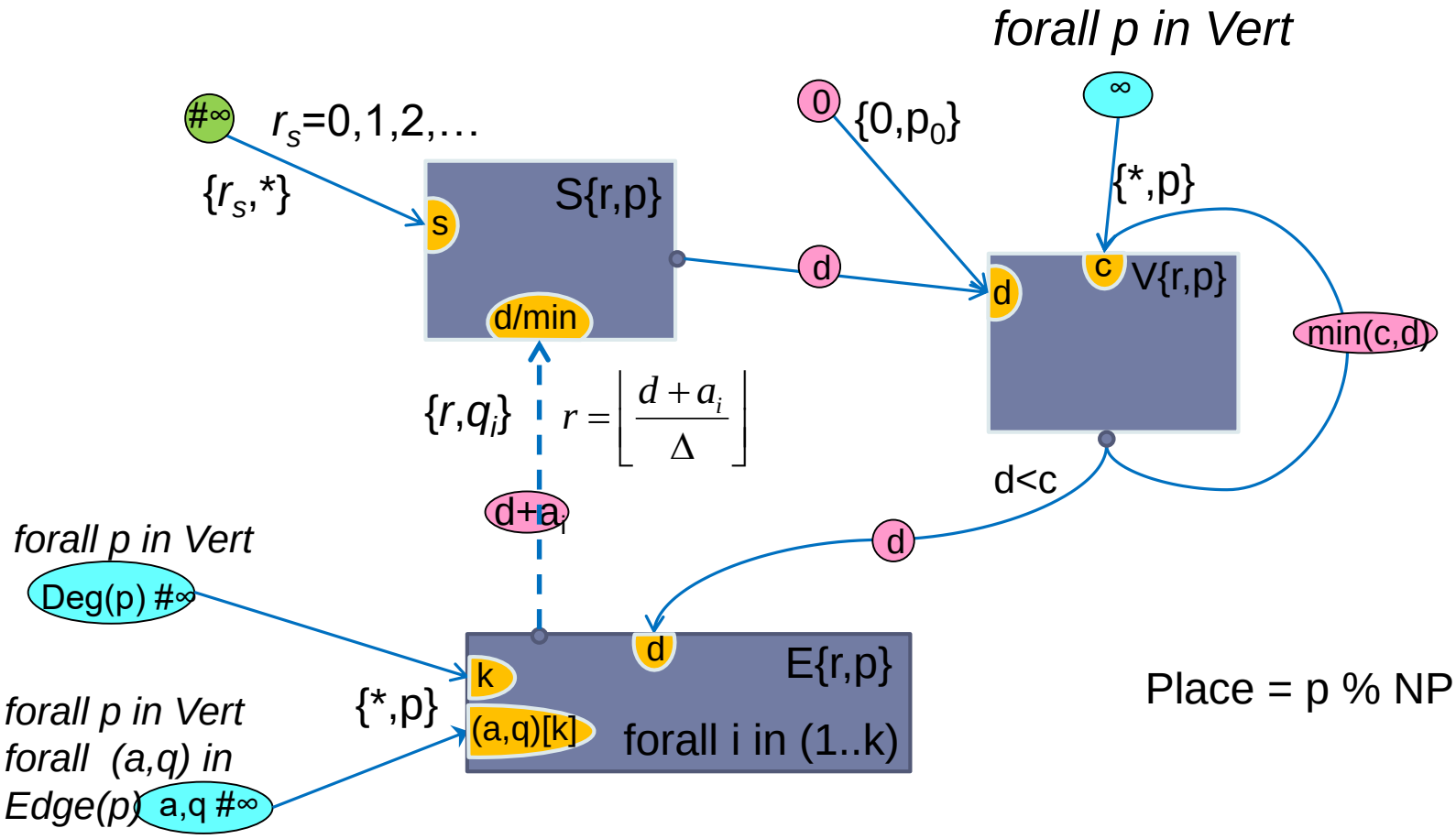
Схема алгоритма SSSP. Наивный вариант (без синхронизации)

Это алгоритм с недетерминированным выполнением, но детерминированным результатом



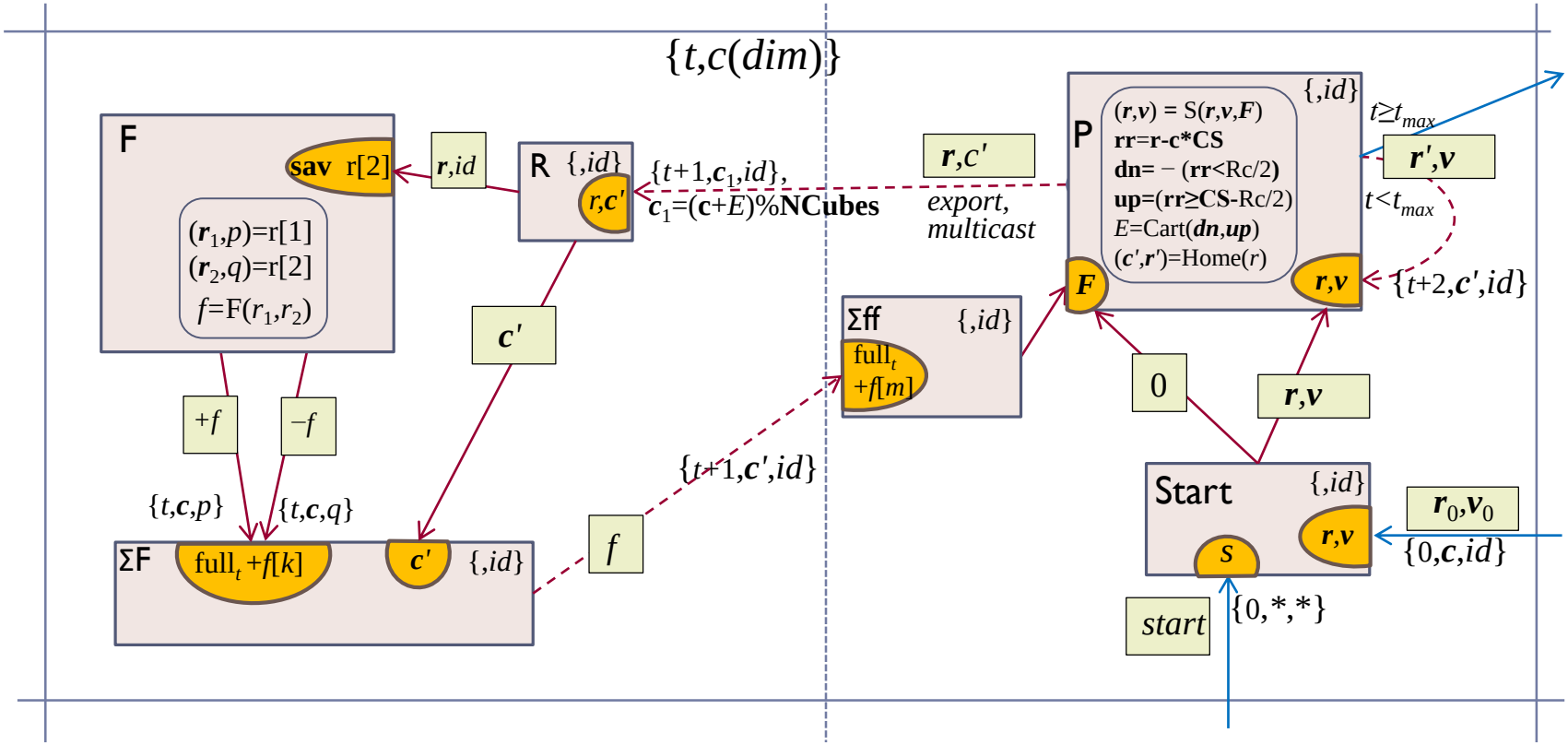
По завершению всякой активности кратчайшие расстояния от p_0 до p, будут находиться в V.c{p} . Возможны многократные проходы по дугам.
Пунктиром обозначены передачи по дугам. Только они могут быть нелокальными.
Узлы распределяются по ядрам по полю p взятием остатка от деления на число ядер NP.

Схема алгоритма SSSP.
Базовый вариант с синхронизацией по Δ-кольцам



Окружение реализует *многоуровневый детектор тишины,* который порождает токен ()⟶V.s{*r*+w,*}, когда на уровне *r* фиксируется завершение активности (w – ширина окна активности).
В начале работы создаются w таких токенов для r=0,1…,w-1. Для стандартного варианта w=1.
На входе V.d выполняется редукция через *min.*

АСИНХРОННАЯ МОЛЕКУЛЯРНАЯ ДИНАМИКА
ПОТОКОВЫЙ АЛГОРИТМ ОРГАНИЗАЦИИ ВЗАИМОДЕЙСТВИЙ



Обозначения:

+n[m] – суммирующий вход, **full**$_t$ - число слагаемых заранее не известно и определяется условием, что должны пройти все (например, по тишине). Здесь индекс t – номер раунда определения тишины.

$\boldsymbol{c}'$ – координаты кубоида, в котором будет главная копия частицы (home)

dn(***up***) – вектор из компонент -1 или 0 (соответственно, 0 или 1) – нижние (верхние) границы 3D-интервала направлений экспорта

$\boldsymbol{c}_1$– множество направлений экспорта $\boldsymbol{c}+\mathbf{dn} \le \boldsymbol{c}_1 \le \boldsymbol{c}+\mathbf{up}$

Cart – операция взятия декартова произведения - вектора интервалов, например, Cart((-1,0,-1), (1,1,0)) имеет 12 элементов (параллелепипед 3x2x2)

Узлы:

F – вычисляет силу взаимодействия пары

ΣF – суммирует силы в рамках кубоида

Σff– суммирует парциальные силы

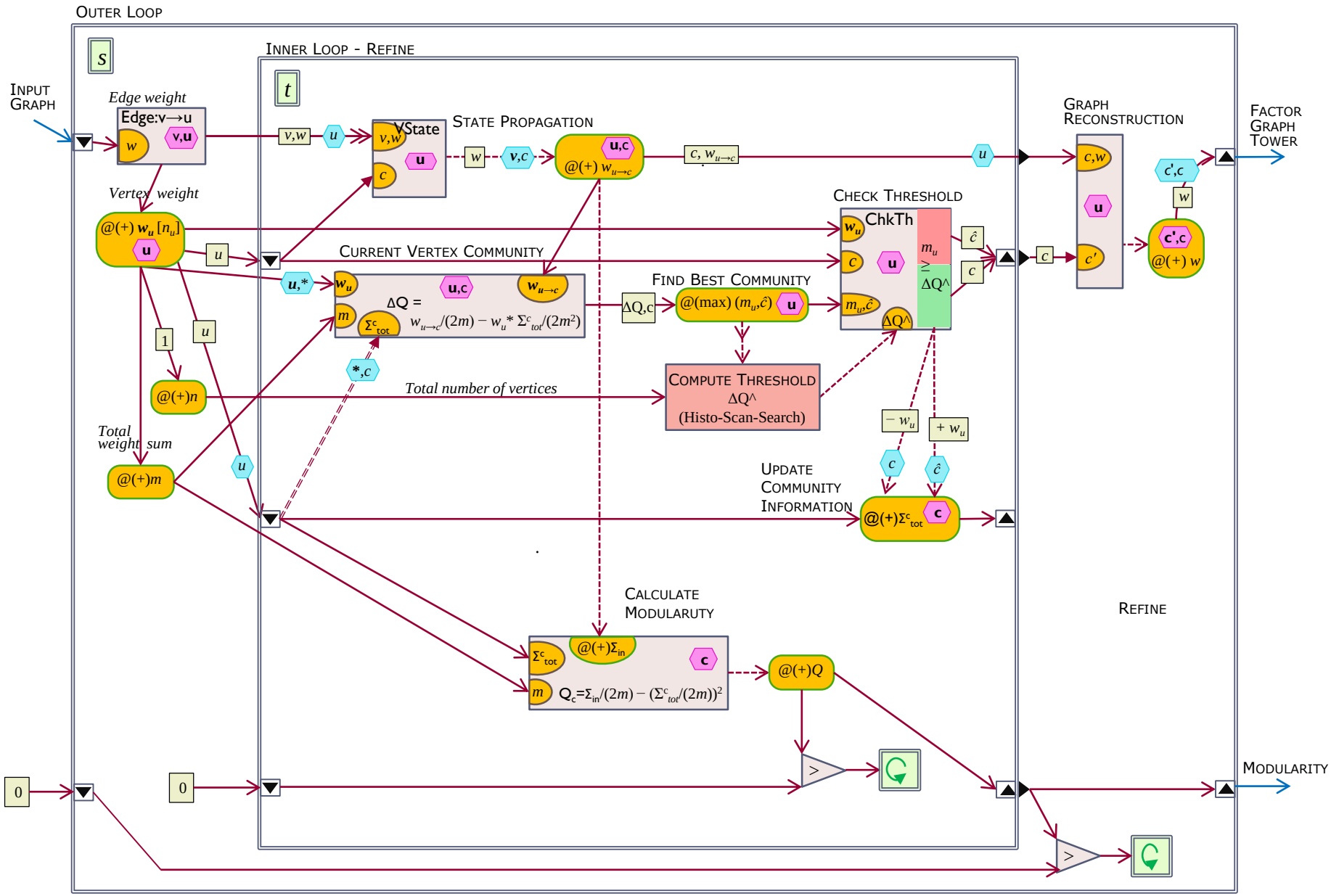
P – вычисляет новые координаты и скорости (***r***,***v***), посылая ее соседям $\boldsymbol{c}_1$ и на новое место $\boldsymbol{c}'$

R – подача координат на **F** и обратного адреса на **ΣF**

Start – инициация процесса при t=0

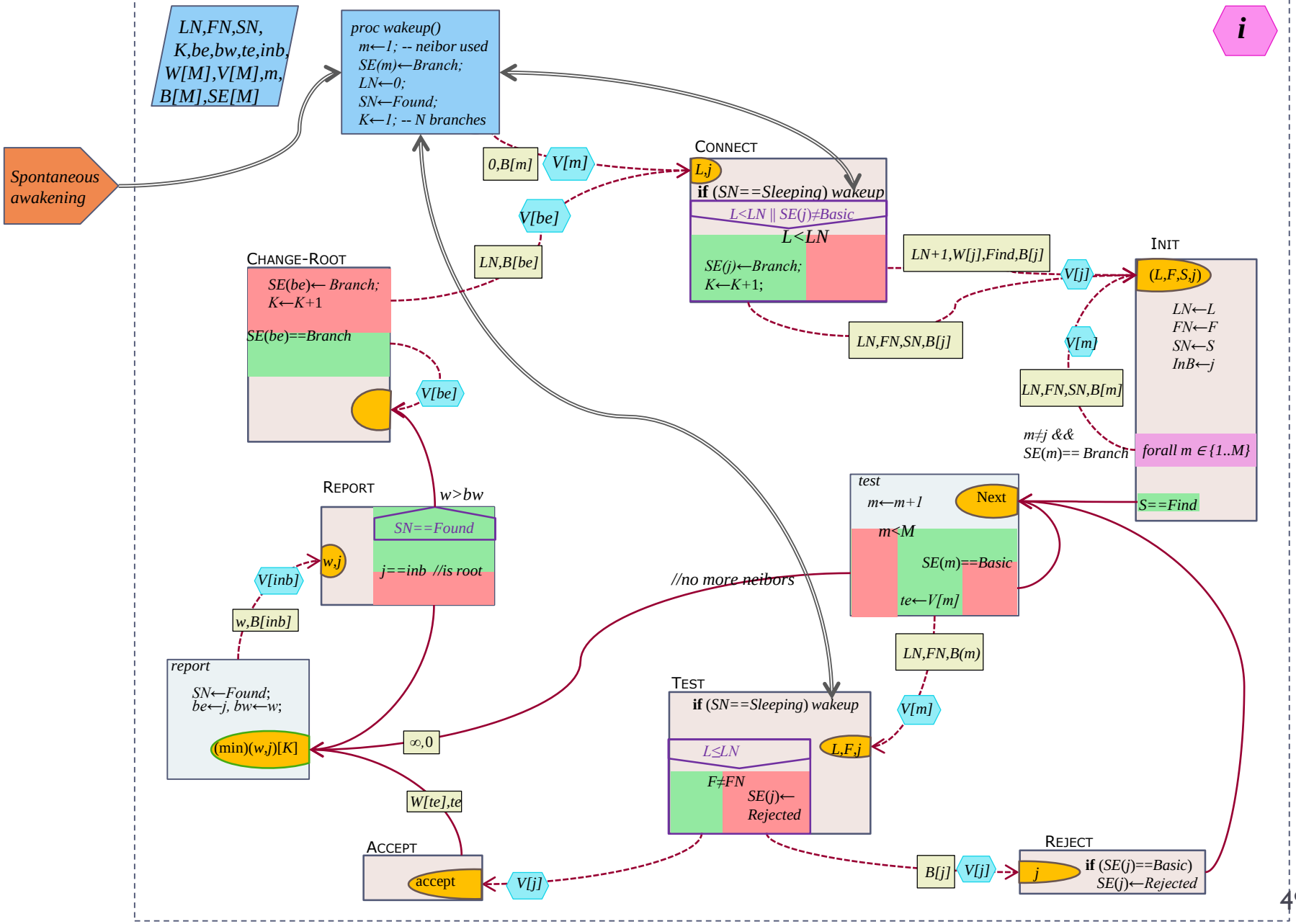
The Louvain Algorithm Flowgraph

according to Q. Xinyu et.al. Scalable Community Detection with the Louvain Algorithm, IPDPS-2015, May, 2015.



GHS algorithm: Distributed minimum spanning tree

(without mutual reject optimization)



Описание алгоритма задачи N тел на обычном математическом языке

$F_{tij} = m_i d_{tij} / \left\| d_{tij} \right\|^3$	Действие тела j на тело i в момент времени t
$d_{tij} = r_{ti} - r_{tj}\ , \quad i \neq j$	Положение тела i относительно тела j в момент времени t
$r_{t+1,i} = r_{ti} + v_{ti}^{+} \Delta t$	Положение тела i в момент времени t+1
$v_{ti}^{+} = v_{ti} + \Delta v_{ti}$	Скорость тела i в момент времени i+1/2
$v_{ti} = v_{ti}^{-} + \Delta v_{ti}$	Скорость тела i в момент времени i
$v_{t,i+1}^{-} = v_{ti}^{+}$	Скорость тела i в момент времени i+1/2
$a_{ti} = \sum_{i \neq j} F_{tij}$	Действие на тело i всех других тел в момент времени t
$\Delta v_{ti} = a_{ti}\ (G/2) \Delta t$	Половинное приращение скорости тела i в момент времени t G – гравитационная постоянная
Дано: $m_i, r_{0i}, v_{0i}, t > 0$	Массы и начальные положения и скорости всех тел
Найти: r_{ti}, v_{ti}	Положения и скорости всех тел в момент времени t

Такой перевод возможен при выполнении двух условий:

1. Все стрелки графа задачи обратимы
2. Каждый экземпляр каждого узла при выполнении активируется ровно один раз