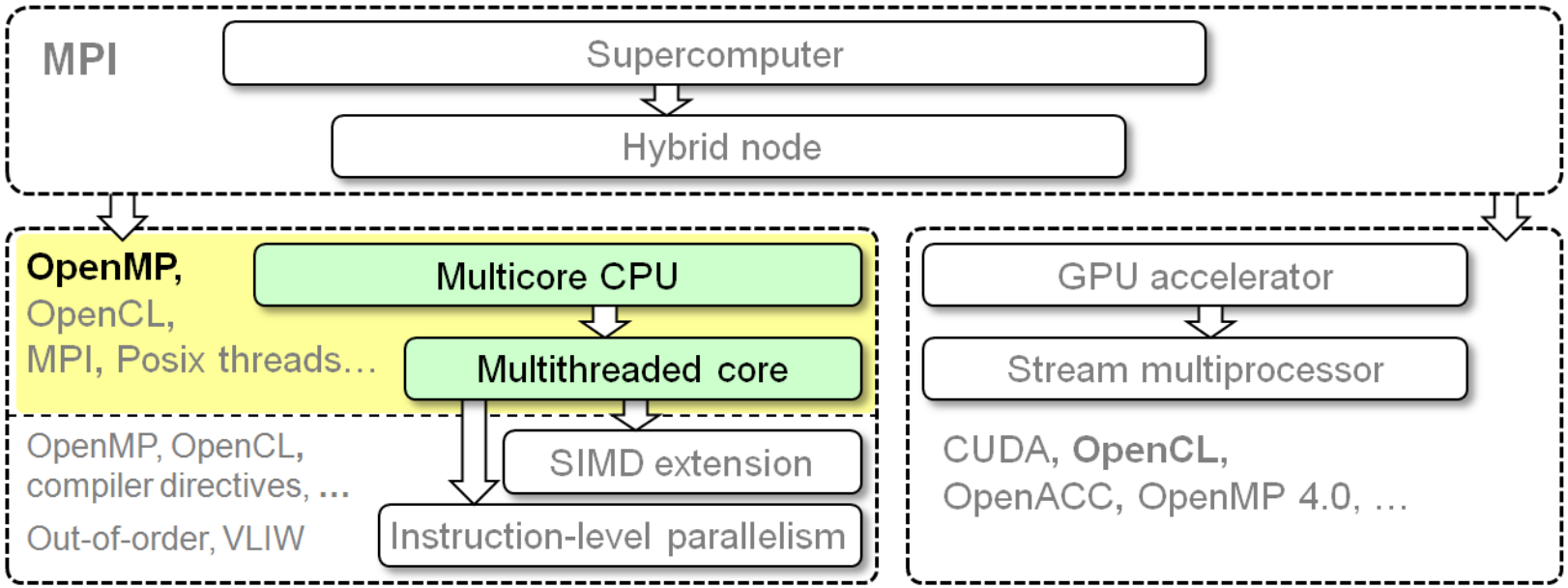
# Параллельные (высокопроизводительные) вычисления

## SM MIMD – многопоточное распараллеливание OpenMP

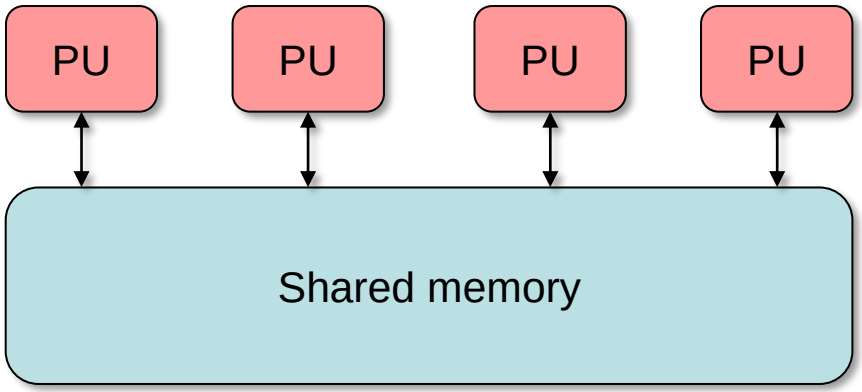


**Сектор вычислительной аэродинамики и аэроакустики**
**ИПМ им. М. В. Келдыша РАН**
http://caa.imamod.ru

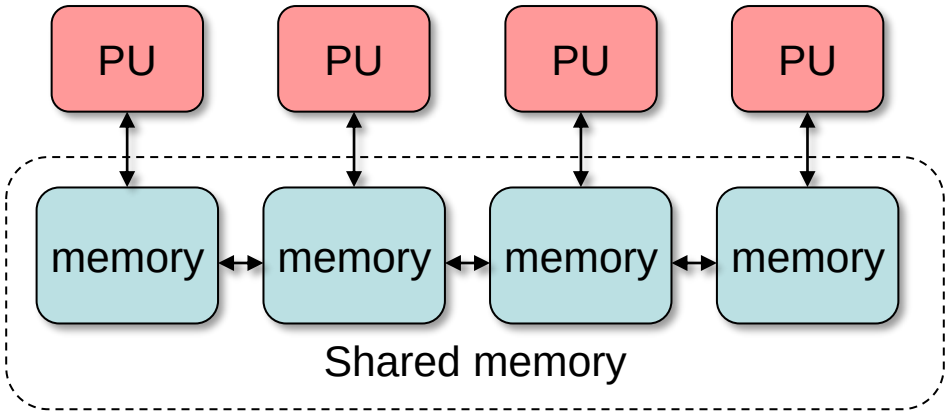
# SM MIMD


MPI
Supercomputer
Hybrid node
OpenMP,
OpenCL,
MPI, Posix threads…
Multicore CPU
Multithreaded core
SIMD extension
Instruction-level parallelism
OpenMP, OpenCL,
compiler directives, …
Out-of-order, VLIW
GPU accelerator
Stream multiprocessor
CUDA, OpenCL,
OpenACC, OpenMP 4.0, …

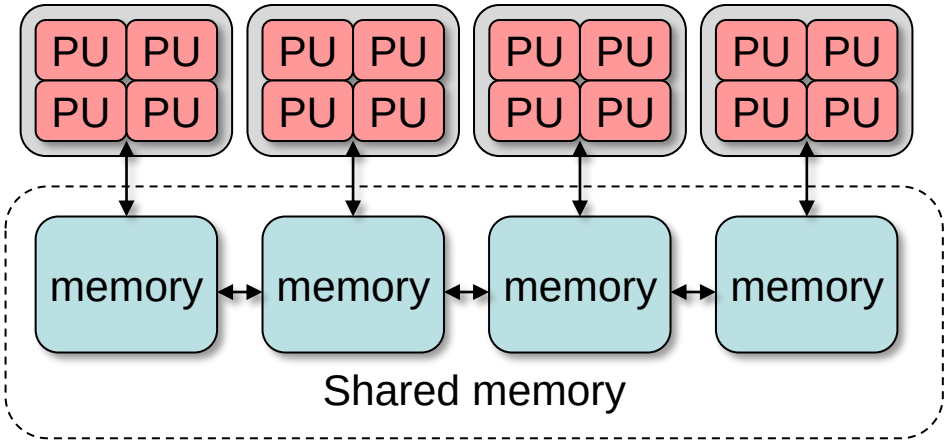
# SM MIMD



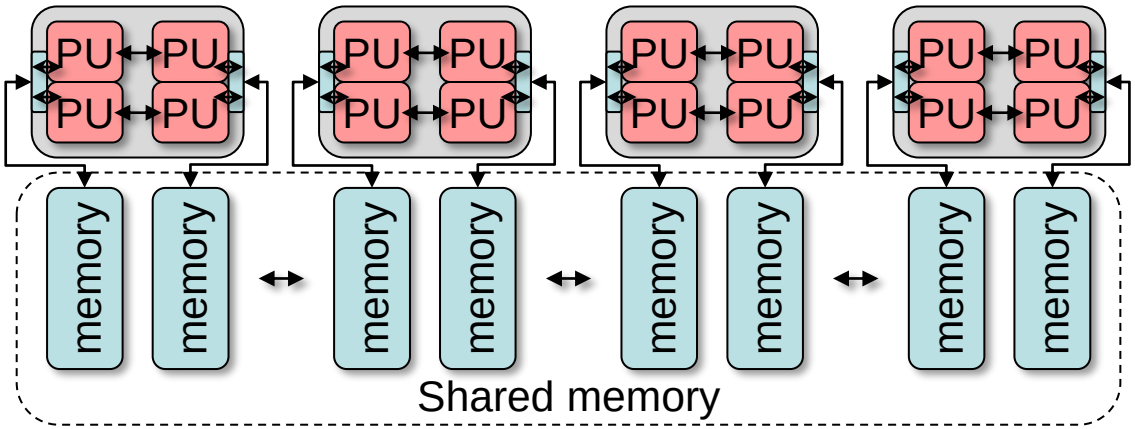
# SM MIMD



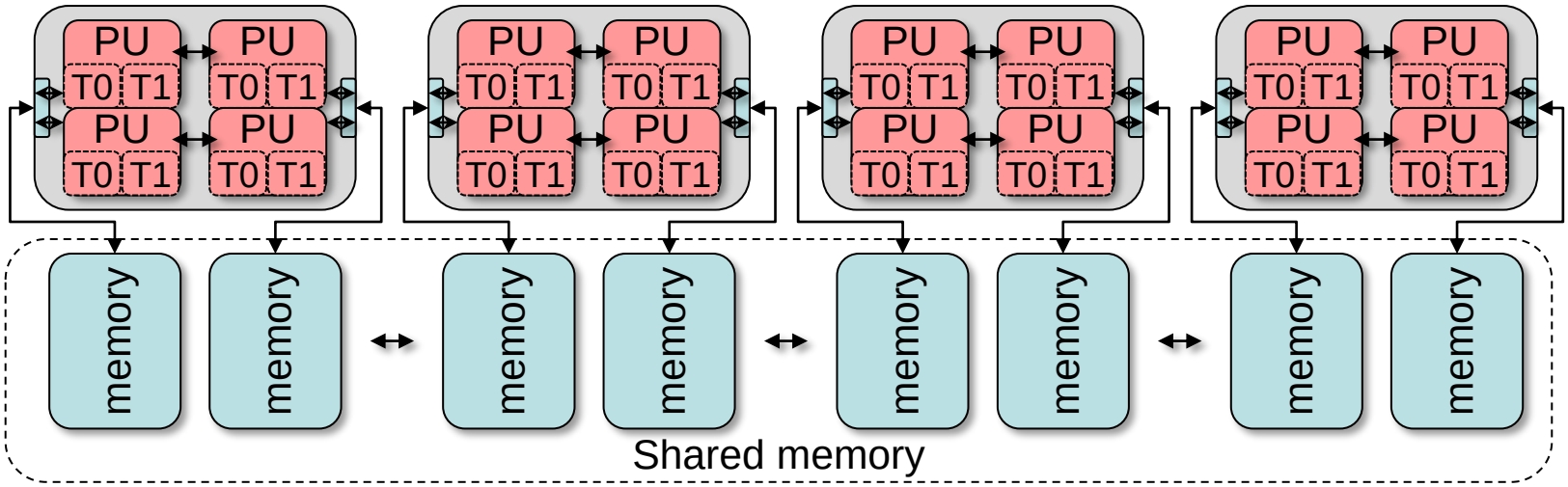
# SM MIMD


PU PU
PU PU
PU PU
PU PU
PU PU
PU PU
PU PU
PU PU
memory
memory
memory
memory
Shared memory

# SM MIMD


PU
PU
PU
PU
PU
PU
PU
PU
PU
PU
PU
PU
PU
PU
PU
PU
memory
memory
memory
memory
memory
memory
memory
memory
Shared memory

# SM MIMD



# Директивное средство для многопоточного распараллеливания библиотечная часть API + директивы

```
#include <omp.h>

omp_set_num_threads(8); // библиотечный вызов
// выставляем число нитей в будущих параллельных областях (например, 8 нитей)
#pragma omp parallel // директива - открытие параллельной области
// для следующего за ней блока
{ // начало параллельной области
    #pragma omp for // директива - параллельный цикл, итерации распределяются между нитями
    for(int i=0; i<n; ++i) x[i] = x[i]*y[i];
} // закрытие параллельной области


#pragma omp parallel num_threads(8) // указываем напрямую число нитей для данной области
{
    #pragma omp for // директива цикла
    for(int i=0; i<n; ++i) x[i] = x[i]*y[i];
}

#pragma omp parallel for num_threads(8) // совмещенная директива для параллельного цикла
for(int i=0; i<n; ++i) x[i] = x[i]*y[i];

omp_set_num_threads(8);
#pragma omp parallel // директива
{
    const int ntr = omp_get_num_threads(); // библиотечный вызов - узнаем число нитей
    const int trn = omp_get_thread_num(); // узнаем номер данной нити
    const int ibeg = trn*(/*объем работы одной нити*/ n/ntr) +
        (/*раздача остатка, если n не делится на ntr*/ trn < n%ntr ?  trn : n%ntr );
    const int iend = ibeg + (n/ntr) + (trn < n%ntr);
    for(int i=ibeg; i<iend; ++i) x[i] = x[i]*y[i];
}
```

```
double res = 0.0;
for(int i=0; i<n; ++i) res += x[i]*y[i];
```

```
double res = 0.0;
#pragma omp parallel
{
    #pragma omp for // директива цикла
    for(int i=0; i<n; ++i) res += x[i]*y[i];
}
```

## Так сойдет?

```
double res = 0.0;
for(int i=0; i<n; ++i) res += x[i]*y[i];
```

```
double res = 0.0;
#pragma omp parallel
{
    #pragma omp for // директива цикла
    for(int i=0; i<n; ++i) res += x[i]*y[i];
}
```

## Так сойдет?

## Гонка / data  race / race condition / зависимость / пересечение по данным

```
double res = 0.0;
#pragma omp parallel
{
    double lres = 0.0;  // частная частичная сумма данной нити
    #pragma omp for
    for(int i=0; i<n; ++i) lres += x[i]*y[i]; // копим результат в собственную переменную
    // соберем общую сумму, избегая гонки
    const int ntr = omp_get_num_threads(); // узнаем число нитей
    const int trn = omp_get_thread_num(); // узнаем номер данной нити
    for(int i=0; i<ntr; ++i){
        #pragma omp barrier // барьерная синхронизация всех потоков
        if(i == trn) res += lres;
    }
}
```

## Результат будет такой же?
## Результат будет правильный?

```
double res = 0.0;
#pragma omp parallel for reduction(+:res)
for(int i=0; i<n; ++i) res += x[i]*y[i];

double res = 0.0;
#pragma omp parallel
{
    double lres = 0.0;  // частная частичная сумма данной нити
    #pragma omp for
    for(int i=0; i<n; ++i) lres += x[i]*y[i]; // копим результат в собственную переменную
    #pragma omp atomic
    res += lres;
}

double res = 0.0;
#pragma omp parallel
{
    #pragma omp for
    for(int i=0; i<n; ++i)
        #pragma omp atomic
        res += x[i]*y[i];
}

 double res = 0.0;
#pragma omp parallel
{
    double lres = 0.0;  // частная частичная сумма данной нити
    #pragma omp for
    for(int i=0; i<n; ++i) lres += x[i]*y[i]; // копим результат в собственную переменную
    #pragma omp critical
    {
         res += lres;
    }
}
```

```
int i,j;
#pragma omp parallel for
for(i=0; i<n; ++i)
    for(j=0; j<m; ++j)
        w[i][j] += v[i][j];
```

```
int i,j;
#pragma omp parallel for
for(i=0; i<n; ++i)
    for(j=0; j<m; ++j)
        w[i][j] += v[i][j];
```

```
#pragma omp parallel for private(j)
for(i=0; i<n; ++i)
    for(j=0; j<m; ++j)
        w[i][j] += v[i][j];
```

```
int i,j;
#pragma omp parallel for
for(i=0; i<n; ++i)
    for(j=0; j<m; ++j)
        w[i][j] += v[i][j];
```

```
#pragma omp parallel for private(j)
for(i=0; i<n; ++i)
    for(j=0; j<m; ++j)
        w[i][j] += v[i][j];
```

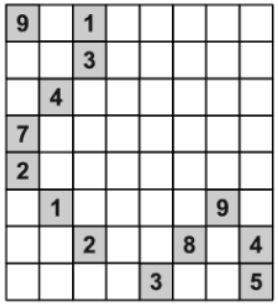
```
#pragma omp parallel for
for(int i=0; i<n; ++i)
    for(int j=0; j<m; ++j)
        w[i][j] += v[i][j];
```

```
#pragma omp parallel for
for(int i=0; i<n; ++i){
    double sum = 0.0;
    const int jb = ia[i];
    const int je = ia[i+1];
    for(int j=jb; j<je; ++j) sum += a[j]*b[ja[j]];
    x[i] = sum;
}

#pragma omp parallel for schedule(dynamic)
for(int i=0; i<n; ++i){
    double sum = 0.0;
    const int jb = ia[i];
    const int je = ia[i+1];
    for(int j=jb; j<je; ++j) sum += a[j]*b[ja[j]];
    x[i] = sum;
}

tbeg = omp_get_wtime();
#pragma omp parallel for schedule(dynamic,100)
for(int i=0; i<n; ++i){
    double sum = 0.0;
    const int jb = ia[i];
    const int je = ia[i+1];
    for(int j=jb; j<je; ++j) sum += a[j]*b[ja[j]];
    x[i] = sum;
}
twcl += omp_get_wtime() – tbeg;
```

## SpMV с матрицей в формате CSR



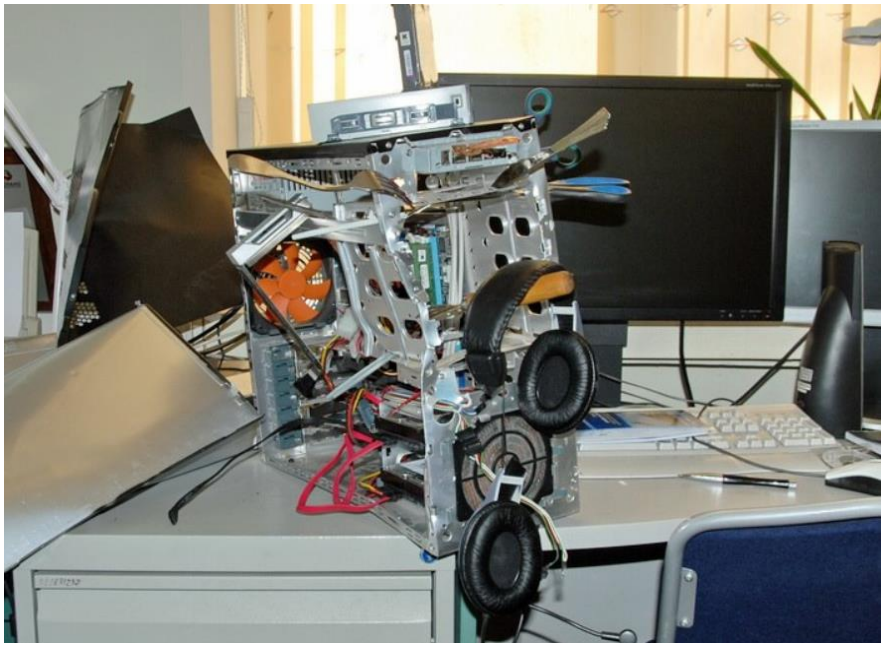

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | 9 | 1 | 3 | 4 | 7 | 2 | 1 | 9 | 2 | 8 | 4 | 3 | 5 |
| JA | 0 | 2 | 2 | 1 | 0 | 0 | 1 | 6 | 2 | 5 | 7 | 4 | 7 |
| IA | 0 | 2 | 3 | 4 | 5 | 6 | 8 | 11 | 13 | | | | |

# Многопоточное распараллеливание

**Средства разработки: OpenMP**, pthreads, TBB, …

**Неприятности:**

- **race condition - пересечение по данным**
  **следствия:**
  расходняк, нестабильное поведение,
  magic bugs, сложная отладка, …

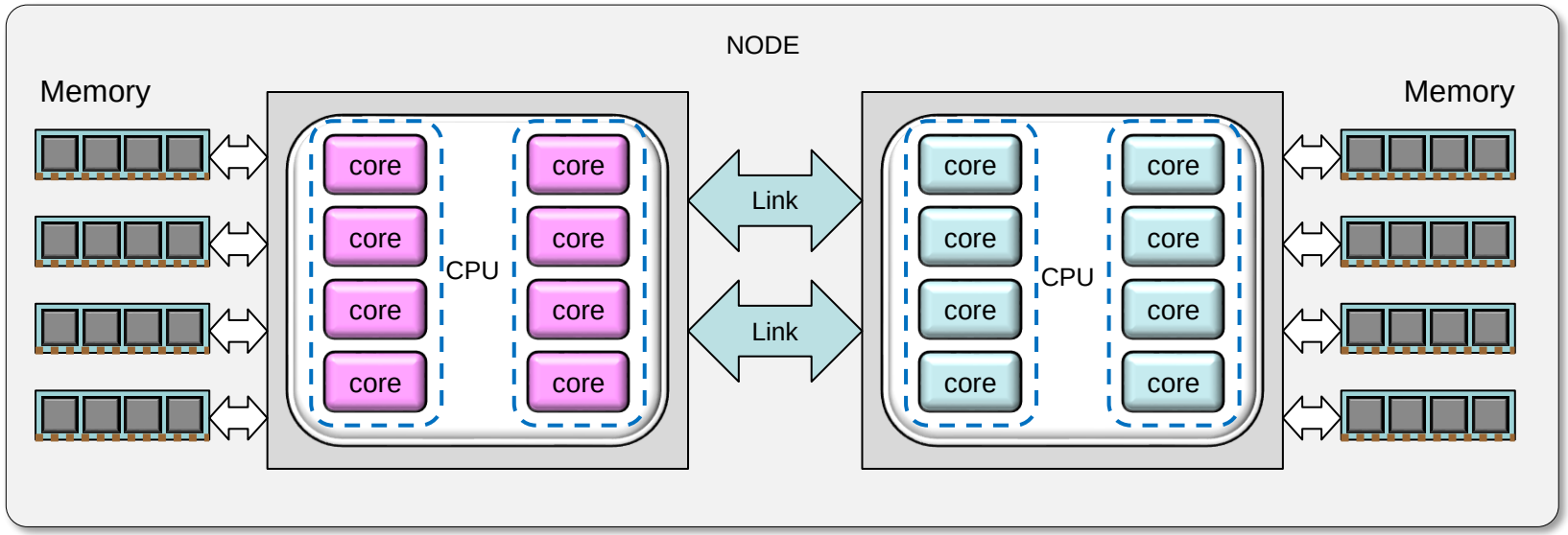


- **Миграция потоков**
- **NUMA эффекты**
- **Дисбаланс**
- **False sharing**

# Накладные расходы

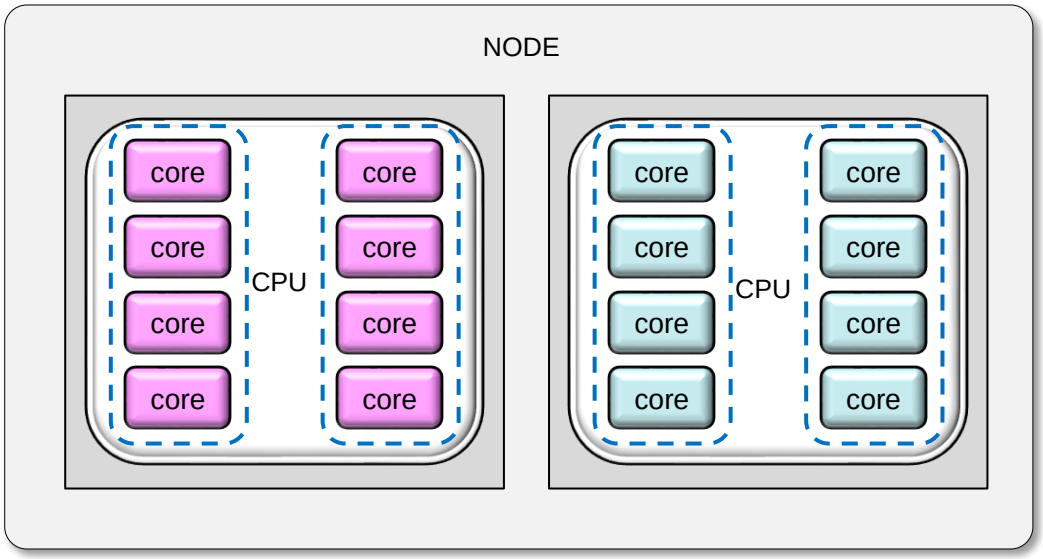
- открытие параллельной области
  особенно с другим числом потоков  (особенно на Phi)
- барьерная синхронизация
- критические секции и атомики
- кэш когерентность

# NUMA между процессорами



# Affinity: привязка потоков к виртуальным процессорам

1 1 1 1  1 1 1 1  0 0 0 0  0 0 0 0




0 0 0 0  0 0 0 0  1 1 1 1  1 1 1 1

# Affinity: привязка потоков к виртуальным процессорам

**Вычислительные ресурсы узла:  виртуальные процессоры (ВП)**

**Их число P = S×C×T,** где
S – число CPU сокетов, C – число ядер в CPU, T – число потоков в ядре

**Битовая маска нити задает для каждого ВП значение 0 или 1:**

**0 – эта нить не может занимать этот процессор**
**1 – эта нить может занимать этот процессор**

**Обычно нумерация такая:**
номер ВП **p =  t*(C*S) + s*C + c,** где
c – номер ядра в CPU, s – номер CPU сокета, t – номер потока в ядре

**Пример – два 6-ядерных CPU с включенным Hyper-threading**

| CPU0 HT0 | CPU1 HT0 | CPU0 HT1 | CPU1 HT1 | |
|---|---|---|---|---|
| 1 1 1 1 1 1 | 1 1 1 1 1 1 | 1 1 1 1 1 1 | 1 1 1 1 1 1 | – можно всё |
| 1 1 1 1 1 1 | 0 0 0 0 0 0 | 1 1 1 1 1 1 | 0 0 0 0 0 0 | – только один CPU |
| 0 0 1 0 0 0 | 0 0 0 0 0 0 | 0 0 1 0 0 0 | 0 0 0 0 0 0 | – только одно ядро |

# Affinity: привязка потоков к виртуальным процессорам

- **Linux**

  **#include <sched.h>**

  - **int sched_setaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask);**
  - **int sched_getaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask);**

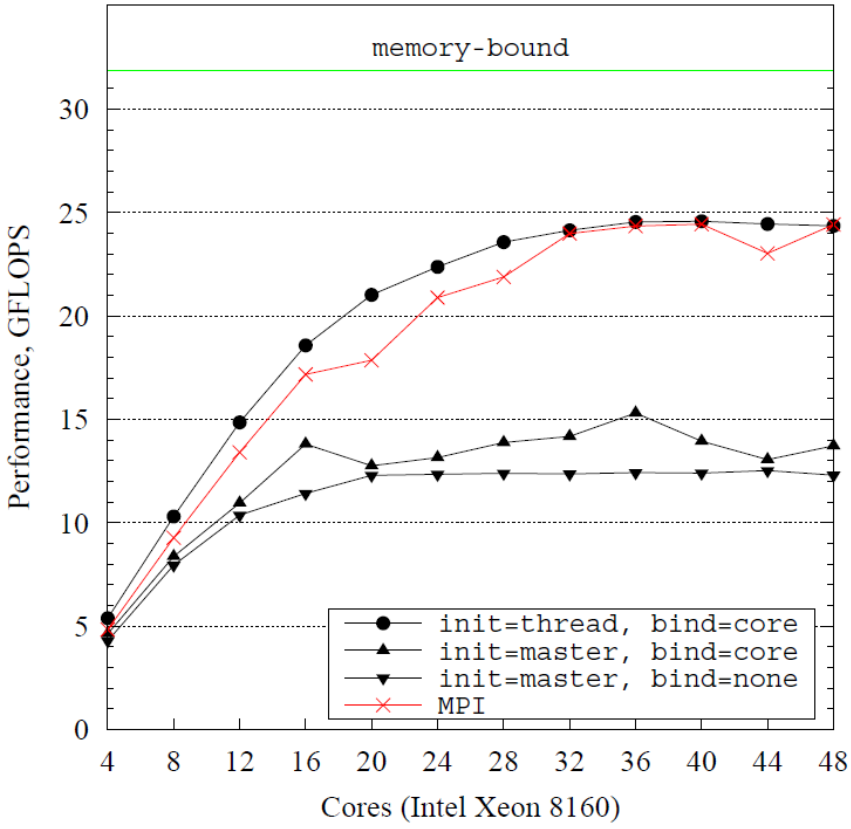
    CPU_ZERO() Clears set, so that it contains no CPUs.
    CPU_SET() Add CPU cpu to set.
    CPU_CLR() Remove CPU cpu from set.
    CPU_ISSET() Test to see if CPU cpu is a member of set.
    CPU_COUNT() Return the number of CPUs in set.

- **Windows**

```
#include <windows.h>
GetCurrentProcess();
GetCurrentThread();
GetProcessAffinityMask(process, &processAffinityMask, &sysmask);
SetProcessAffinityMask(process, processAffinityMask);
SetThreadAffinityMask(thread, threadAffinityMask);
```

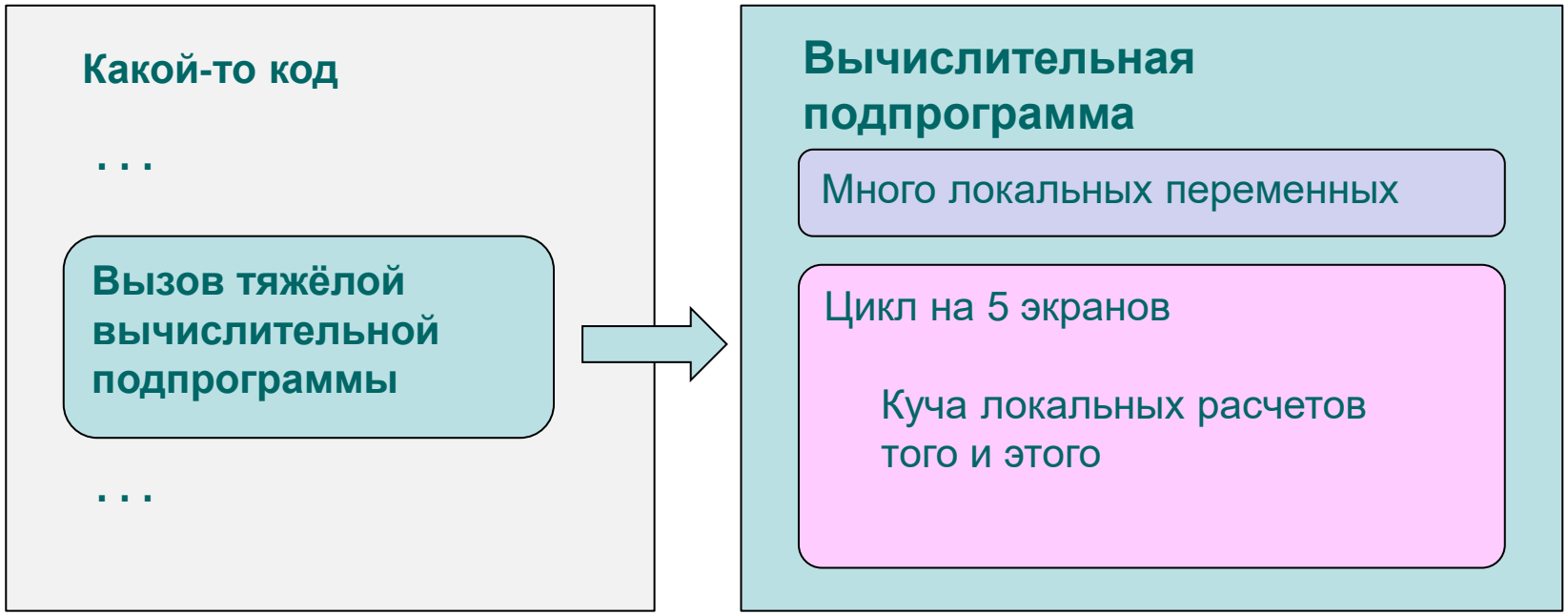
# Доступ к памяти под OpenMP в условиях NUMA

- **First touch rule – кто проинициализировал, того и данные**
- **Одна общая декомпозиция по нитям**
  вместо параллелизма циклов

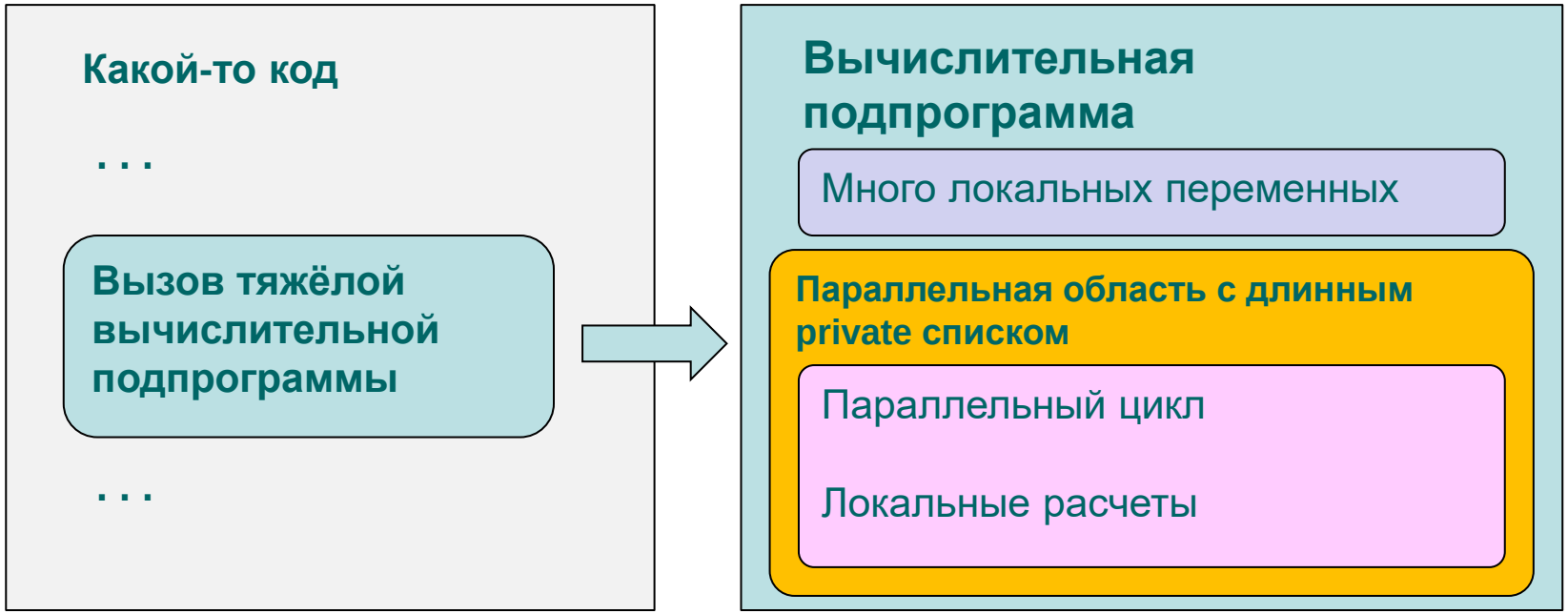



**SpMV на узле 2 x 24C CPU**

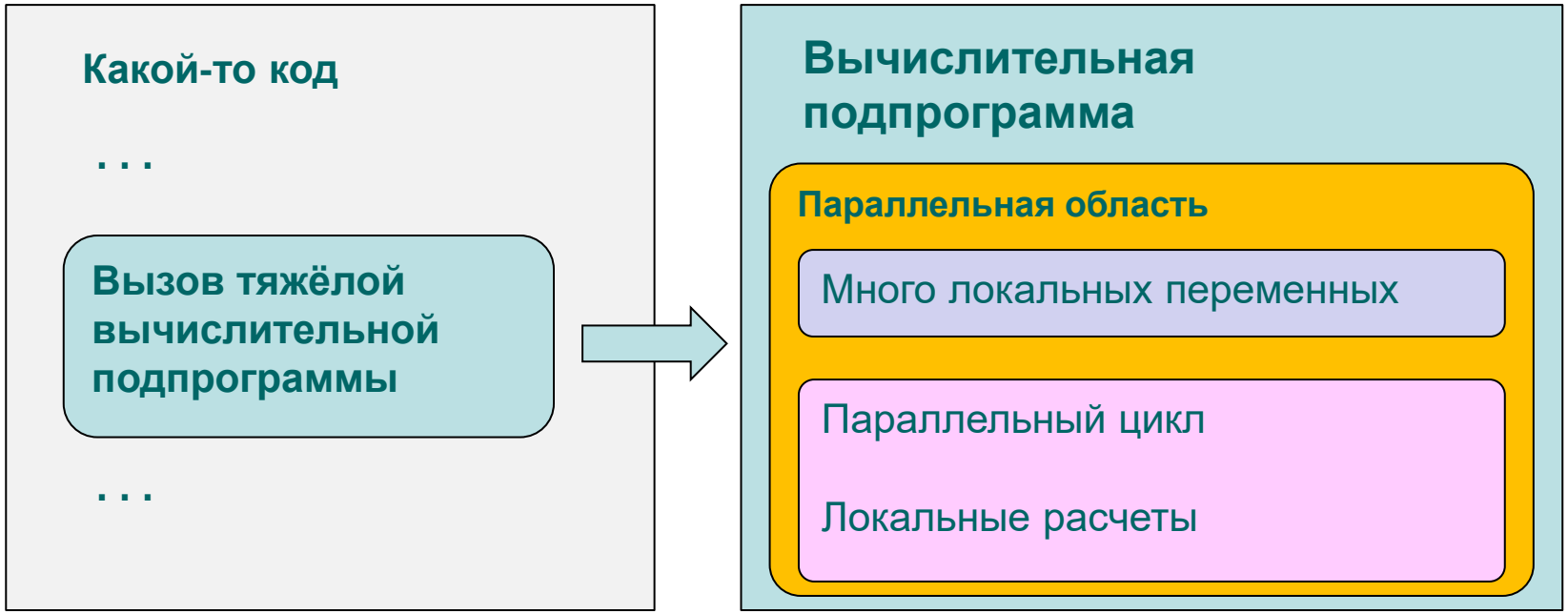
# Реализация распараллеливания



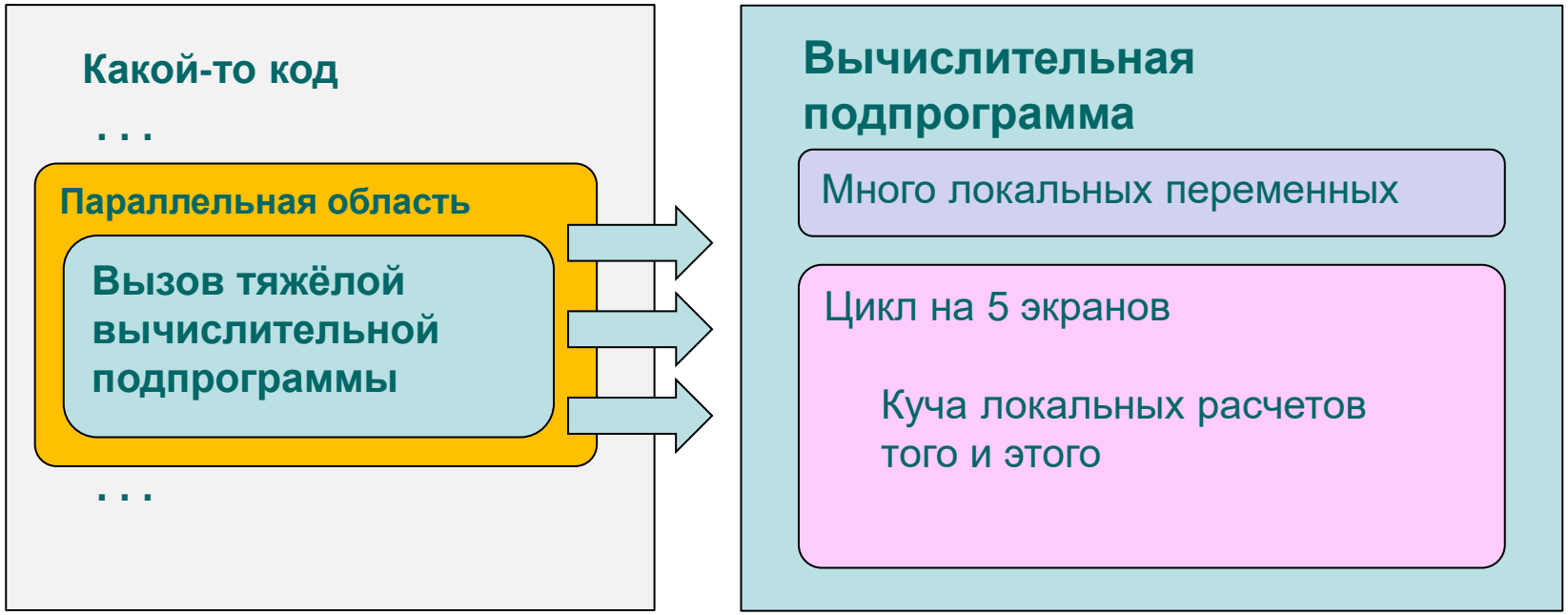
# Реализация распараллеливания



# Реализация распараллеливания



# Реализация распараллеливания



# Параллельная эффективность

- Почему я все идеально распараллелил, а не ускоряется!?

memory-bound? оцениваем TBP

- Почему у меня compute-boumd, а ускоряется не так хорошо?

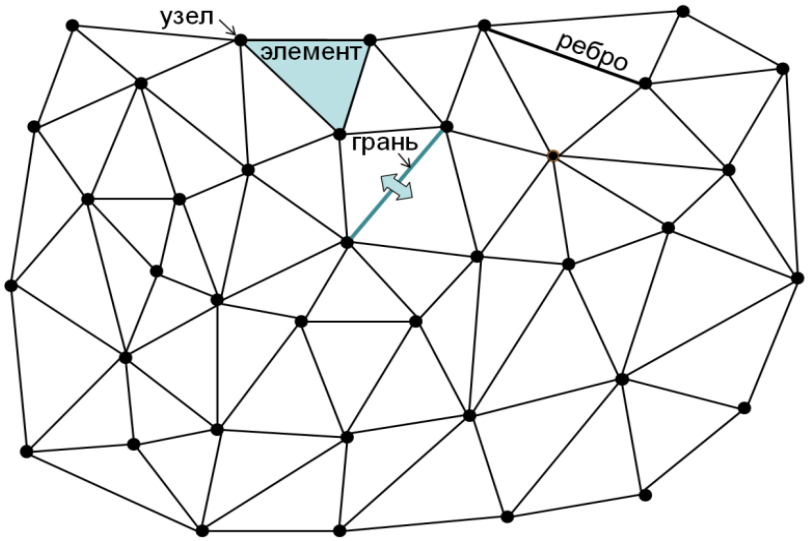
[Modify Frequency Info]

| Mode | Base | Turbo Frequency/Active Cores | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Normal | 2,400 MHz | 3,200 MHz | 3,200 MHz | 3,000 MHz | 3,000 MHz | 2,900 MHz | 2,900 MHz | 2,900 MHz | 2,900 MHz | 2,800 MHz | 2,800 MHz |
| AVX2 | 2,000 MHz | 3,100 MHz | 3,100 MHz | 2,900 MHz | 2,900 MHz | 2,600 MHz | 2,600 MHz | 2,600 MHz | 2,600 MHz | 2,400 MHz | 2,400 MHz |
| AVX512 | 1,200 MHz | 2,900 MHz | 2,900 MHz | 2,200 MHz | 2,200 MHz | 1,700 MHz | 1,700 MHz | 1,700 MHz | 1,700 MHz | 1,600 MHz | 1,600 MHz |

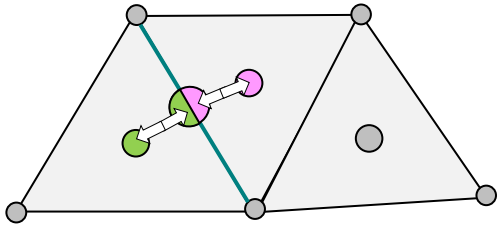
https://en.wikichip.org/wiki/intel/frequency_behavior

# Устранение зависимостей по данным в сеточном методе

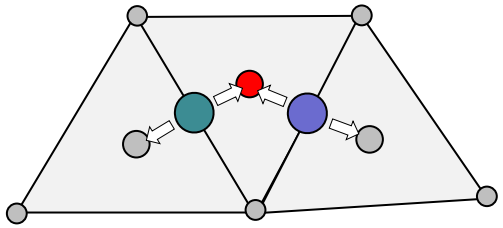
## Операция: расчет потоков через грани ячеек в конечно-объемном CFD алгоритме




Расчет потока через грань



Добавление потока в ячейки



race

```
for(int iface=0; iface<NF/*Number of faces*/; ++iface){ // цикл по граням
    double fluxes[M/*число переменных*/]; // потоки через грань
    // вычисляем потоки через i-ю грань
    CalculateFlux(iface, fluxes); // большая тяжелая функция
    // далее берем из массива граней номера инцидентных ячеек
    // и добавляем потоки в эти ячейки, но с разным знаком
    for(int j=0; j<M; ++j) Var[Faces[iface].cells[0]][j] -= fluxes[j];
    for(int j=0; j<M; ++j) Var[Faces[iface].cells[1]][j] += fluxes[j];
}
```

# Устранение зависимостей по данным в сеточном методе

**Упростим и обобщим. Пусть одна переменная, и не потоки, а просто что-то:**

```
#pragma omp parallel for
for(int ie=0; ie<NE/*Number of edges*/; ++ie){ // цикл по ребрам графа
    double r = Calc(ie, Y); // что-то считаем
    X[E[ie].v[0]] += r; // добавляем что-то
    X[E[ie].v[1]] += r; // для простоты даже пусть с одним знаком
}
```

- **Полное дублирование вычислений (потоков)**
  Цикл по ячейкам, на каждой итерации которого рассчитываются потоки через грани ячейки

```
#pragma omp parallel for
for(int iv=0; iv<NV/*Number of vertices*/; ++iv){ // цикл по вершинам графа
    for(int k=0; k<V[iv].Ne/*степень вершины*/; ++k){ // цикл по ребрам из данной вершины
        int ie = V[iv].e[k]/*e - список ребер из вершины*/; // номер данного ребра
        double r = Calc(ie, Y); // что-то считаем
        X[iv] += r; // добавляем это что-то в вершины
    }
}
```

# Устранение зависимостей по данным в сеточном методе

- **Атомики –** суммирование потоков через атомарные операции

```
#pragma omp parallel for
for(int ie=0; ie<NE/*Number of edges*/; ++ie){ // цикл по ребрам графа
    double r = Calc(ie, Y); // что-то считаем
    #pragma omp atomic
    X[E[ie].v[0]] += r;
    #pragma omp atomic
    X[E[ie].v[1]] += r;
}
```

- **Разделение операции на два этапа**

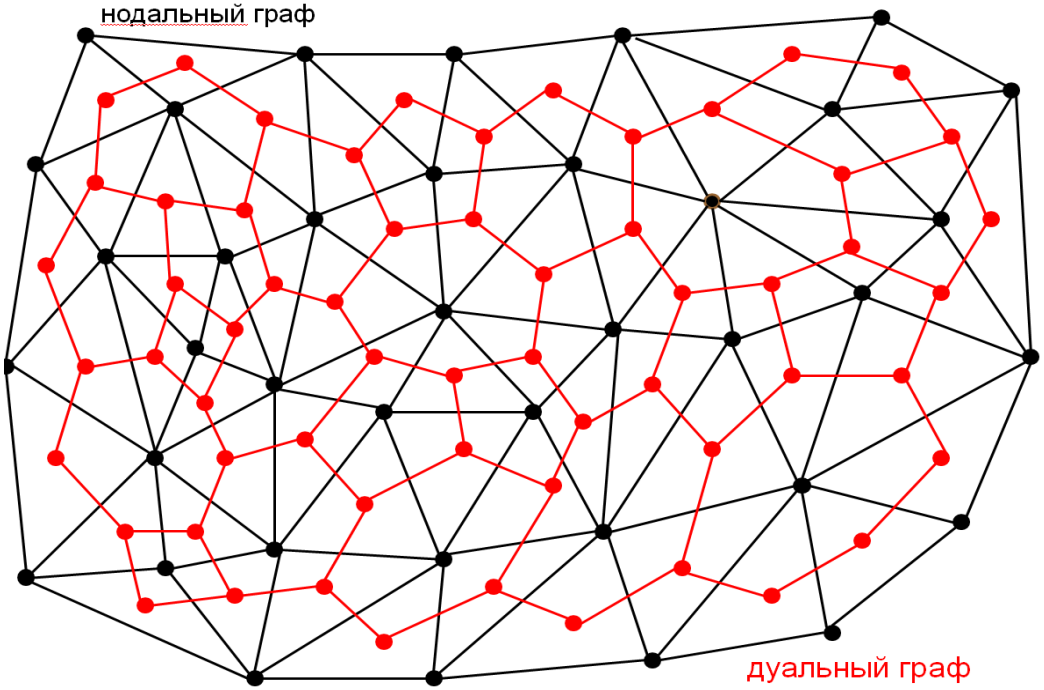
  Цикл по ребрам, результат в промежуточный массив

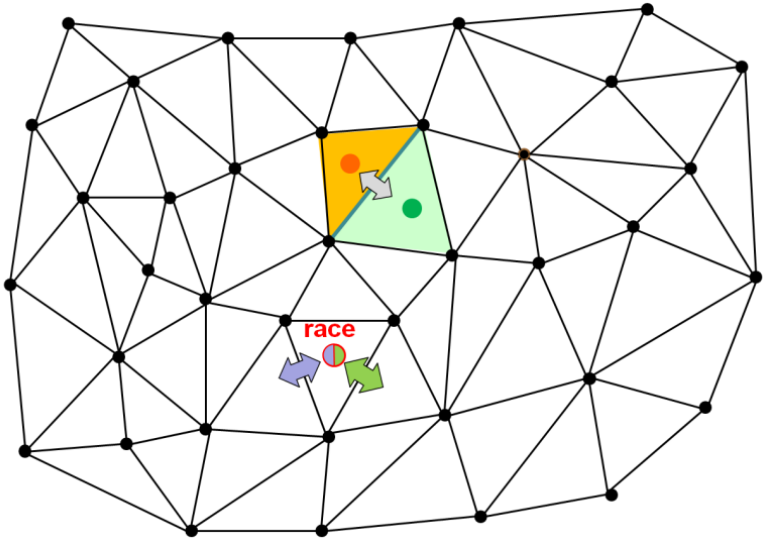
  Суммирование в цикле по вершинам

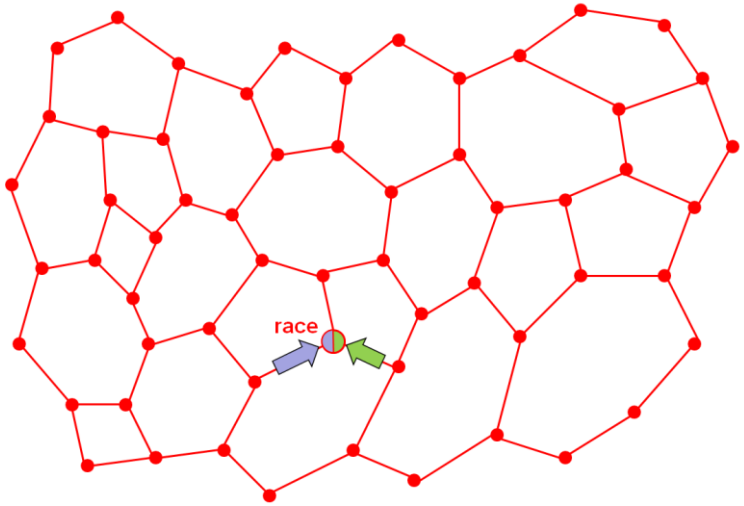
```
#pragma omp parallel
{
    #pragma omp for // первая операция - считаем значения на ребрах
     for(int ie=0; ie<NE/*Number of edges*/; ++ie) // цикл по ребрам графа
         R[ie] = Calc(ie, Y); // что-то считаем, теперь в массив R по всем ребрам
    // #pragma omp barrier – тут должен быть барьер, но он не нужен, у for неявный барьер
    #pragma omp for // вторая операция - добавляем значения с ребер в вершины
    for(int iv=0; iv<NV/*Number of vertices*/; ++iv){ // цикл по вершинам графа
        for(int k=0; k<V[iv].Ne; ++k){ // цикл по ребрам из данной вершины
            int ie = V[iv].e[k]; // номер данного ребра в массиве всех ребер
            X[iv] += R[ie]; // добавляем это что-то в вершину
        }
    }
}
```

# Устранение зависимостей по данным в сеточном методе

**Мы говорим сетка,**
**подразумеваем –**
**граф**

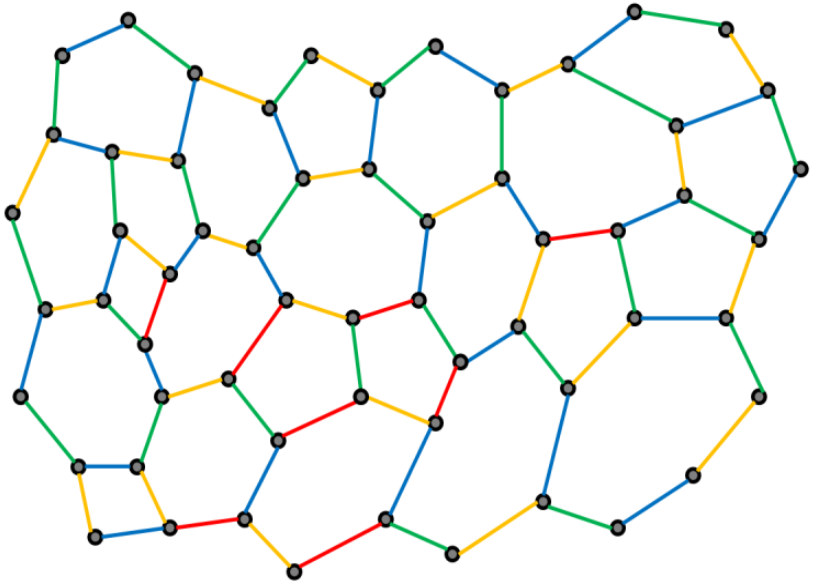


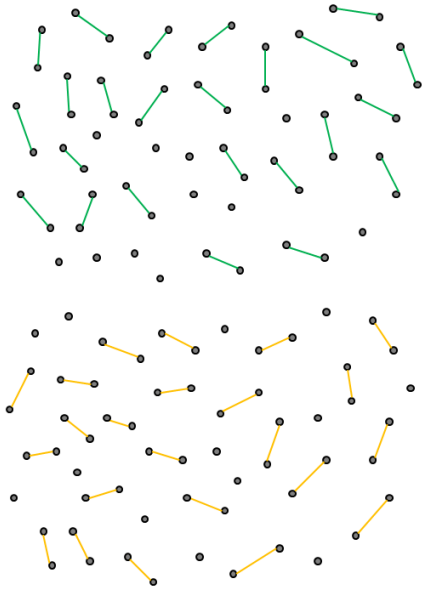



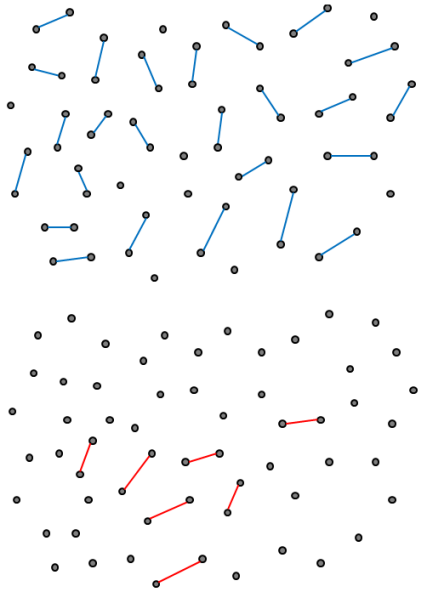



# Устранение зависимостей по данным в сеточном методе

- **Раскраска графа**





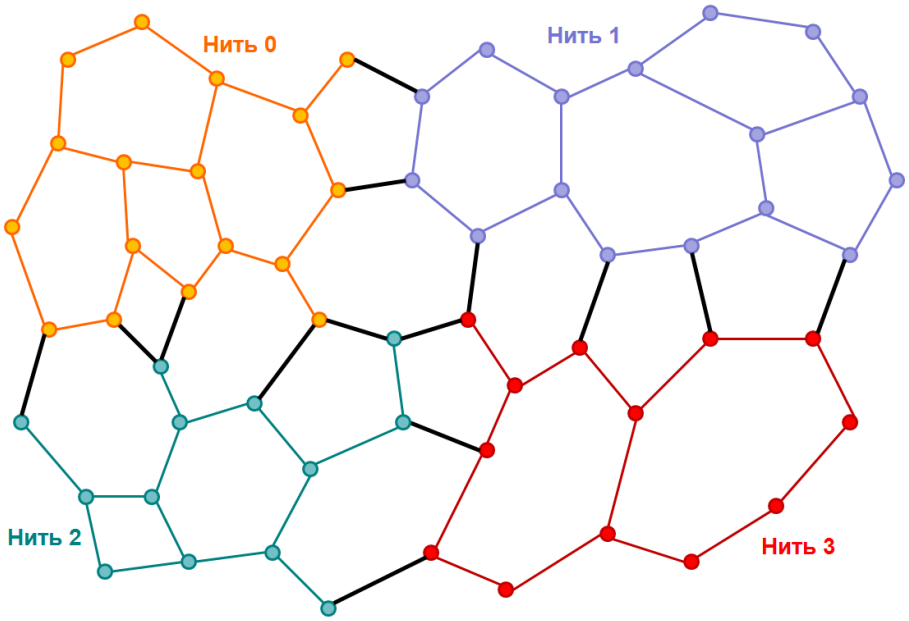


```
#pragma omp parallel
{
    for(int ic=0; ic<NC/*Number of colors*/; ++ic){ // цикл по цветам
        #pragma omp for
        for(int ie=C[ic]; ie<C[ic+1]; ++ie){ // цикл по ребрам графа данного цвета
            double r = Calc(ie, Y);
            X[E[ie].v[0]] += r;
            X[E[ie].v[1]] += r;
        }
       //#pragma omp barrier – тут должен быть барьер, но он не нужен, у for неявный барьер
    }
}
```

# Устранение зависимостей по данным в сеточном методе

- **Декомпозиция. Последовательная обработка интерфейса**

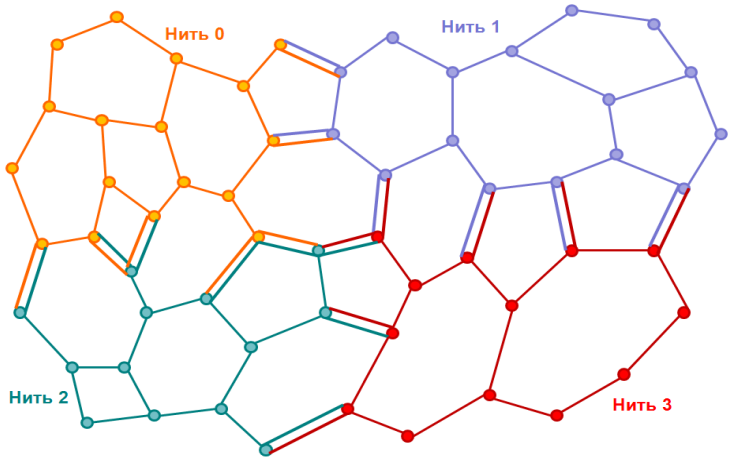
```
#pragma omp parallel
{
    int trn = omp_get_thread_num(); // номер данной нити
    int ntr = omp_num_threads ();   // число нитей
    // теперь тут нет директивы цикла, у каждой нити просто свой диапазон
    for(int ie=C[trn]; ie<C[trn+1]; ++ie){ // цикл по ребрам данной нити – не параллельный!
        double r = Calc(ie, Y);
        X[E[ie].v[0]] += r;
        X[E[ie].v[1]] += r;
    }
    #pragma omp barrier // обязательно барьер! иначе интерфейс налезет и будет гонка
    if(trn == ntr-1){ // последняя нить займется интерфейсом
        for(int ie=C[ntr]; ie<C[ntr+1]; ++ie){ // цикл по интерфейсным ребрам
            double r = Calc(ie, Y);
            X[E[ie].v[0]] += r;
            X[E[ie].v[1]] += r;
        }
    }
}
```



# Устранение зависимостей по данным в сеточном методе

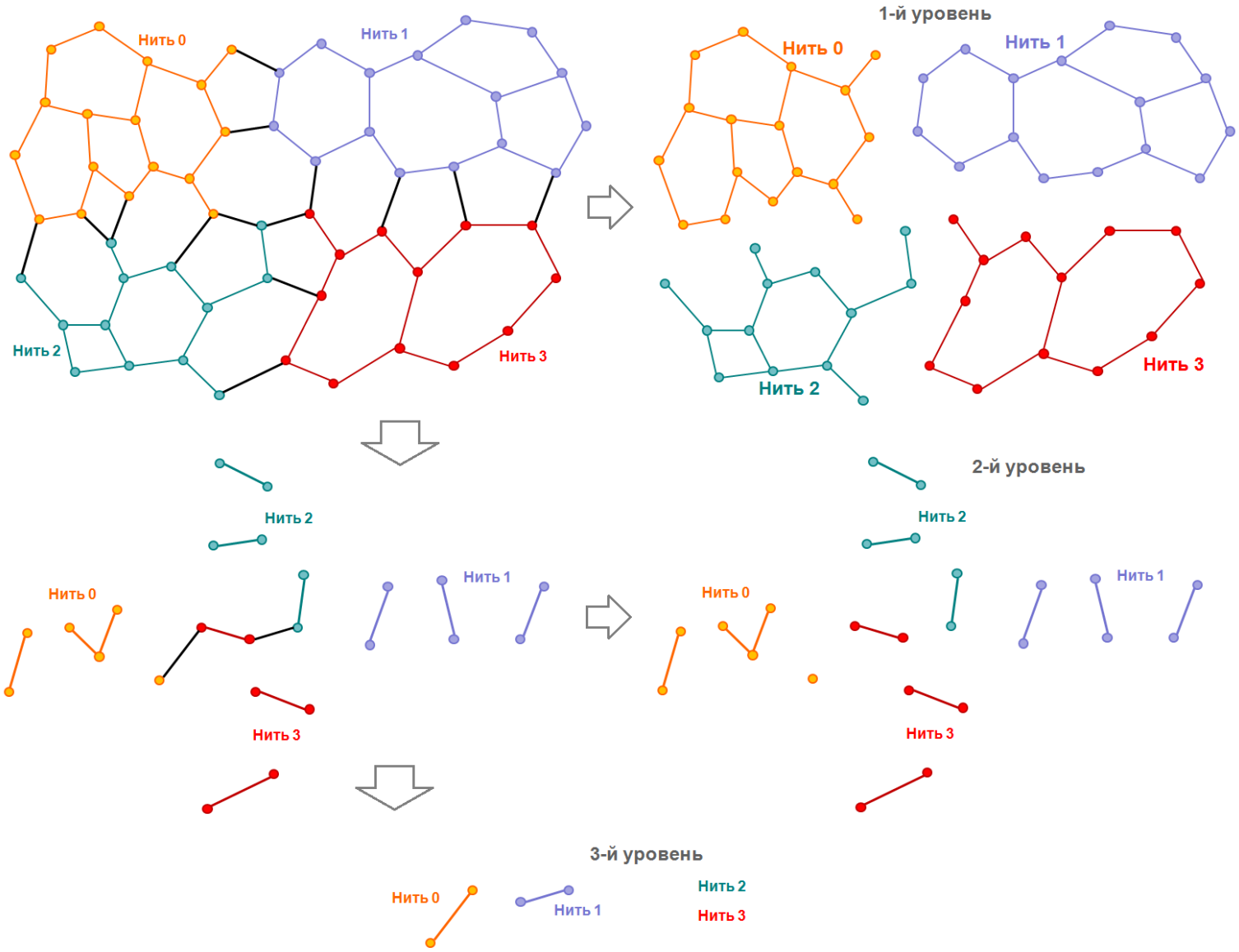
- **Декомпозиция. Дублирование вычислений по интерфейсу**

```
#pragma omp parallel
{
    int trn = omp_get_thread_num(); // номер данной нити
    int ntr = omp_num_threads ();   // число нитей
    // теперь тут нет директивы цикла, у каждой нити просто свой диапазон
    for(int ie=C[trn]; ie<C[trn+1]; ++ie){ // цикл по ребрам данной нити – не параллельный!
        double r = Calc(ie, Y);
        X[E[ie].v[0]] += r;
        X[E[ie].v[1]] += r;
    }
    #pragma omp barrier // обязательно барьер! иначе интерфейс налезет и будет гонка
    { // все нити займутся интерфейсом
        for(int ie=C[ntr]; ie<C[ntr+1]; ++ie){ // цикл по интерфейсным ребрам
            // отмечаем, за какие вершины отвечает эта нить
            bool doit[2] = {P[E[ie].v[0]]==trn, P[E[ie].v[1]]==trn };
            if(!doit[0] && !doit[1]) continue; // наших вершин нет - пропускаем
            double r = Calc(ie, Y);
            X[E[ie].v[doit[0]?0:1]] += r;
        }
    }
}
```



# Устранение зависимостей по данным в сеточном методе

- **Многоуровневая декомпозиция**



# Устранение зависимостей по данным в сеточном методе

- **Многоуровневая декомпозиция**

Упорядочим ребра уровням, внутри каждого уровня – по цветам

```
#pragma omp parallel
{
    int trn = omp_get_thread_num(); // номер данной нити
    for(int il=0; il<NL; ++il){ // цикл по уровням декомпозиции интерфейса
        int *C = L[il]; // берем диапазоны цветов данного уровня
        for(int ie=C[trn]; ie<C[trn+1]; ++ie){ // цикл по ребрам этой нити на этом уровне
            double r = Calc(ie, Y);
            X[E[ie].v[0]] += r;
            X[E[ie].v[1]] += r;
        }
        #pragma omp barrier // обязательно барьер! иначе будет гонка
    }
}
```

# Ссылки

## Стандарт OpenMP

https://www.openmp.org/specifications/

https://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf
https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf

## Учебные материалы

https://parallel.ru/info/parallel/openmp/

https://doi.org/10.1007/978-3-319-98833-7 (глава 3)

https://www.openmp.org/resources/tutorials-articles/

https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf

https://computing.llnl.gov/tutorials/openMP/

## Директивы и функции, которые нужно знать

#pragma omp…
- parallel
- for
- critical
- barrier
- master
- atomic

omp_set_num_threads
omp_get_thread_num
omp_get_num_threads
omp_get_wtime
omp_get_num_procs