Моделирование алгоритмов оптимизации выполнения критических секций на выделенных процессорных ядрах в многоядерных вычислительных системах

Пазников А.А.[1,3], Павский К.В.[1],
Павский В.А.[2], Куприянов М.С.[3]
apaznikov@gmail.com

[1] Институт физики полупроводников им. А.В. Ржанова Сибирского отделения Российской академии наук

[2] Кемеровский технологический институт пищевой промышленности

[3] Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В.И.Ульянова (Ленина)

II Международная научная конференция по проблемам управления в технических системах (ПУТС-2017)

25-27 октября 2017 г.
г. Санкт-Петербург
СПбГЭТУ «ЛЭТИ»

Многоядерные ВС с общей памятью

QuickPath / Hyper-Transport

NUMA-сегмент 1

Контроллер памяти

Кэш-память

L3

L2 L2 L2 L2

L1 L1 L1 L1

Проц. ядра

C1 C2 C3 C4

Кэш-память

L3

L2 L2 L2 L2

L1 L1 L1 L1

Проц. ядра

C5 C6 C7 C8

L3

L2 L2 L2 L2

L1 L1 L1 L1

C1 C2 C3 C4

Кэш-память

Проц. ядра

L3

L2 L2 L2 L2

L1 L1 L1 L1

C1 C2 C3 C4

Кэш-память

Проц. ядра

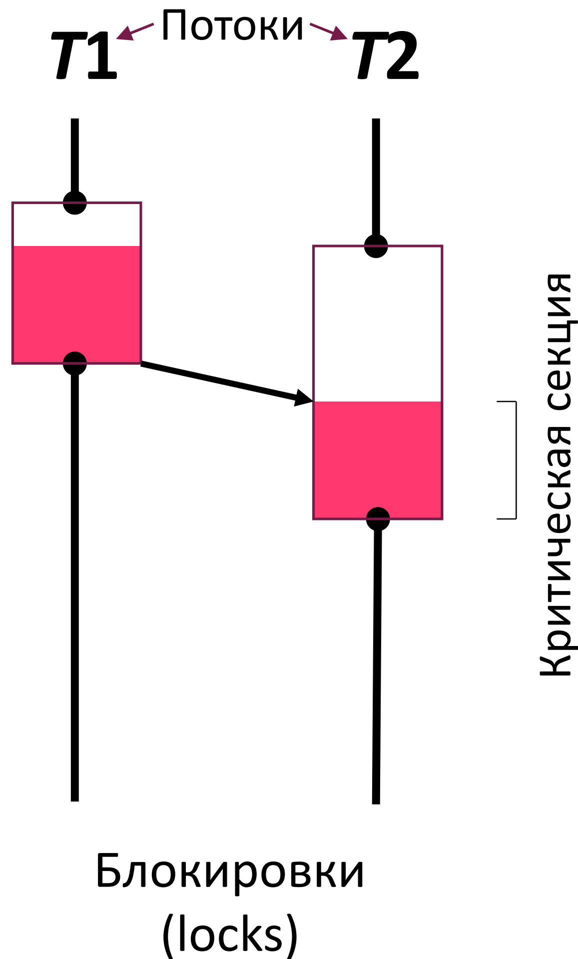
Контроллер памяти

NUMA-сегмент 2

Архитектурные особенности современных многоядерных вычислительных систем (ВС):

- Большемасштабность: количество процессорных ядер в системе – до 10^2 – 10^3
- Иерархическая структура: многоуровневая кэш-память, логические процессоры.
- Неоднородный доступ к памяти (NUMA-системы)
- Гетерогенная структура: наличие специализированных ускорителей и сопроцессоров
- Многообразие механизмов согласованного состояния памяти.

Средства синхронизации многопоточных программ



Блокировки (locks)

Параллельное выполнение критических секций сериализуется – «заменяется» на последовательное.

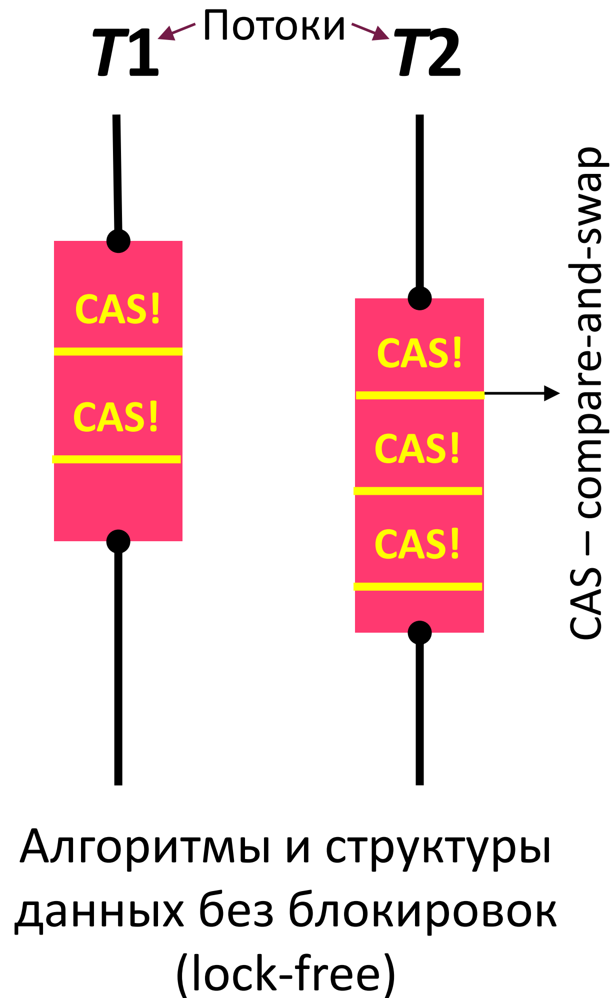
Недостатки:

- **Низкая масштабируемость (scalability)**
- Сериализация (serialization)
- Взаимные блокировки (deadlocks, livelocks)
- Голодание (starvation), возникновение очередей (convoying)
- Инверсии приоритетов (priority inversion)
- Отсутствие очерёдности выполнения критических секций (FIFO)
- Умеренные накладные расходы

Реализации:

- Queue locks (CLH, MCS)
- Spinlocks (test-and-set-based, exponential backoff-based)
- Flat combining (CC-Synch, DSM-Synch, Oyama lock, etc)
- Futex-based (PThreads mutex, PThread read-write mutex)

Средства синхронизации многопоточных программ



Алгоритмы и структуры данных без блокировок (lock-free)

Алгоритмы и структуры данных, свободные от блокировок (lock-free)

Потокобезопасность обеспечивается применением атомарных операций (CAS – compare-and-swap, LL/SC – load link, store conditional).

Недостатки:

- **Высокая трудоёмкость разработки параллельных программ.**
- Проблема освобождения памяти (проблема ABA).
- Неприменимость атомарных операций для переменных большого размера.
- Низкая эффективность выполнения атомарных операций.

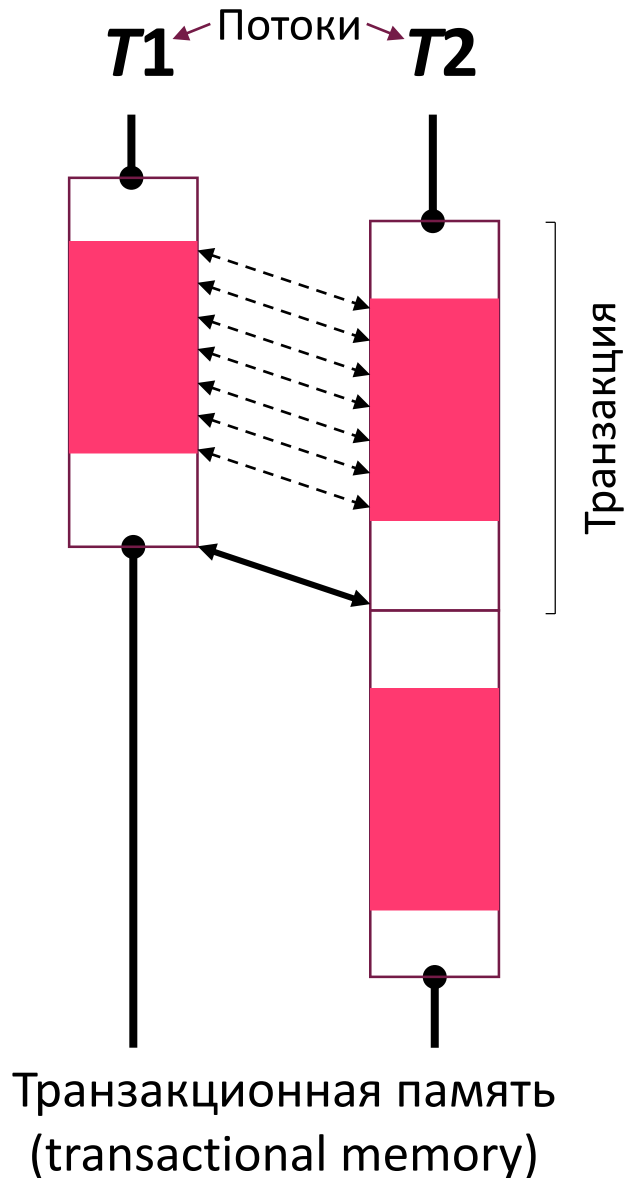
Перспективные алгоритмы:

- Lock-free producer-consumer
- Exponential backoff
- Elimination arrays
- Diffraction trees
- Sorting networks
- Cliff Click hash table
- Skip-list
- Split-ordering

Решение проблемы ABA:

- Quiescent-based schemes
- Pointer-based schemes (hazard pointers, drop-the-anchor, pass-the-buck)
- Reference counting
- Tagged state reference
- Intermediate nodes
- TM-based

Средства синхронизации многопоточных программ



Транзакционная память (transactional memory, TM)

Организация в программе транзакционных секций, в рамках которые реализуется защита разделяемых областей памяти (а не участка кода).

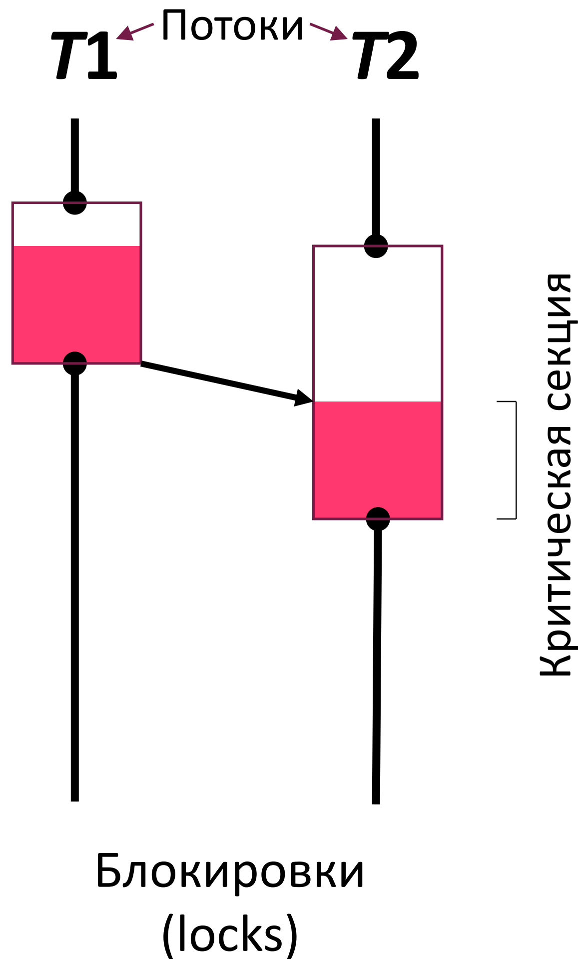
Недостатки:

- **Очень высокие накладные расходы**
- Возможность отмены транзакций
- Ограничения на операции внутри транзакционных секций
- Сложность отладки
- Необходимость переработки программы

Реализации:

- GCC TM
- LazySTM
- TinySTM
- DTMC
- RSTM
- STM Monad

Средства синхронизации многопоточных программ



Блокировки (locks)

Блокировки (locks)

Параллельное выполнение критических секций сериализуется – «заменяется» на последовательное.

Недостатки:

- **Низкая масштабируемость (scalability)**
- Сериализация (serialization)
- Взаимные блокировки (deadlocks, livelocks)
- Голодание (starvation)
- Инверсии приоритетов (priority inversion)
- Отсутствие очерёдности выполнения критических секций (FIFO)
- Умеренные накладные расходы

Реализации:

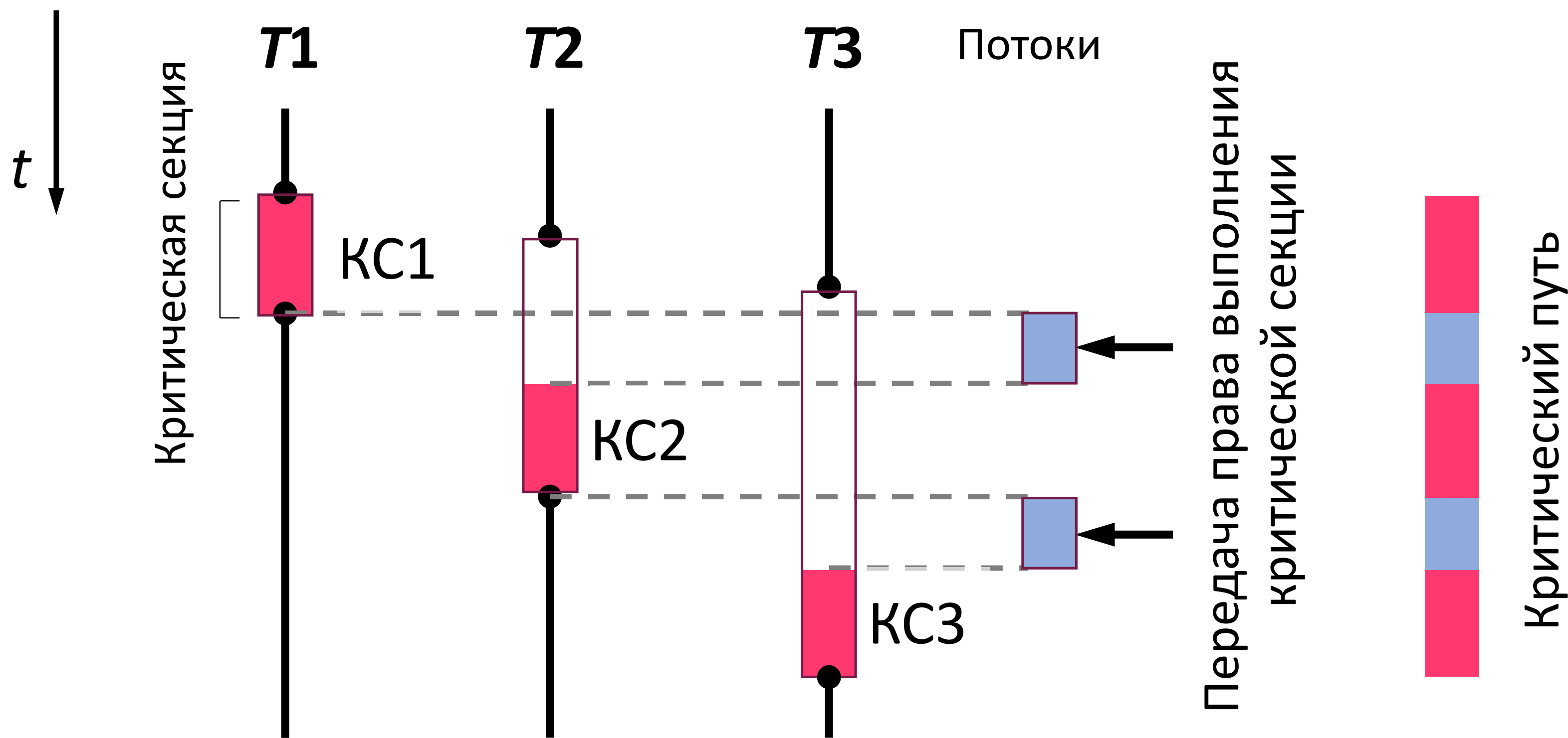
- Queue locks (CLH, MCS)
- Spinlocks (test-and-set-based, exponential backoff-based)
- Flat combining (CC-Synch, DSM-Synch, Oyama lock, etc)
- Futex-based (PThreads mutex, PThread read-write mutex)

- **Наличие узких мест (bottlenecks)**
- **Высокий уровень конкурентного доступа (contention) ⇒ дорогостоящий захват блокировок**

⇓

Необходима разработка более эффективных алгоритмов блокировки!

Время выполнения критической секции



Время выполнений критической секции (критический путь):

$$t = t_1 + t_2,$$

где

t_1 – время выполнения инструкций критической секции,

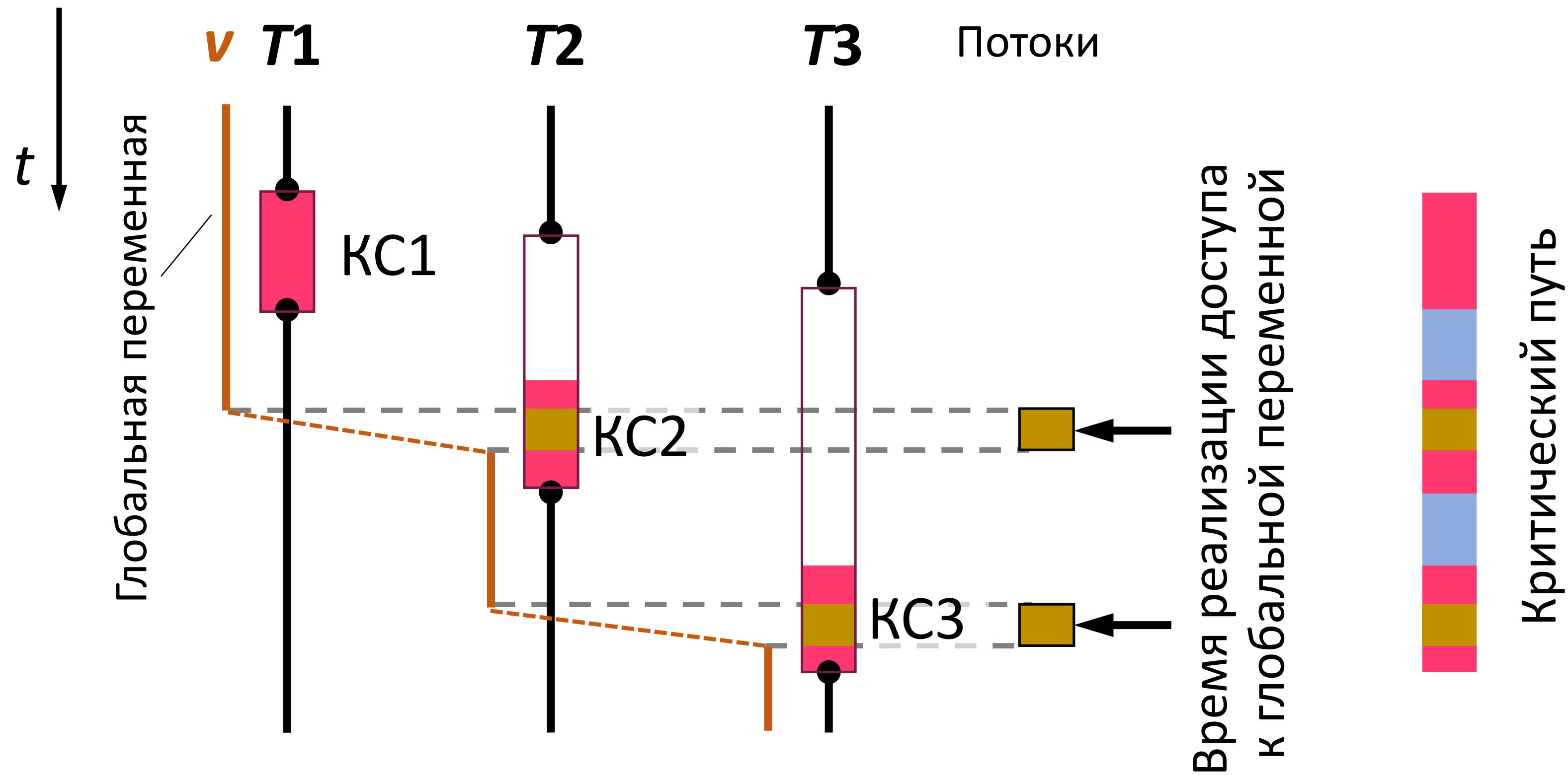
t_2 – время передачи права выполнения критической секции

В существующих алгоритмах блокировки время передачи права выполнения:

- Spinlock $\Rightarrow$ Обращение к глобальному флагу
- PThread mutex $\Rightarrow$ переключение контекста
- MCS $\Rightarrow$ активация потока
- Flat combining $\Rightarrow$ захват глобальной блокировки

Требуется разработка алгоритмов блокировки, позволяющего сократить время передачи права выполнений критической секции.

Время выполнения критической секции



Время выполнений критической секции (критический путь):

$$t = t_1 + t_2 + \boldsymbol{t_3},$$

где

t_1 – время выполнения инструкций критической секции,

t_2 – время передачи права выполнения критической секции,

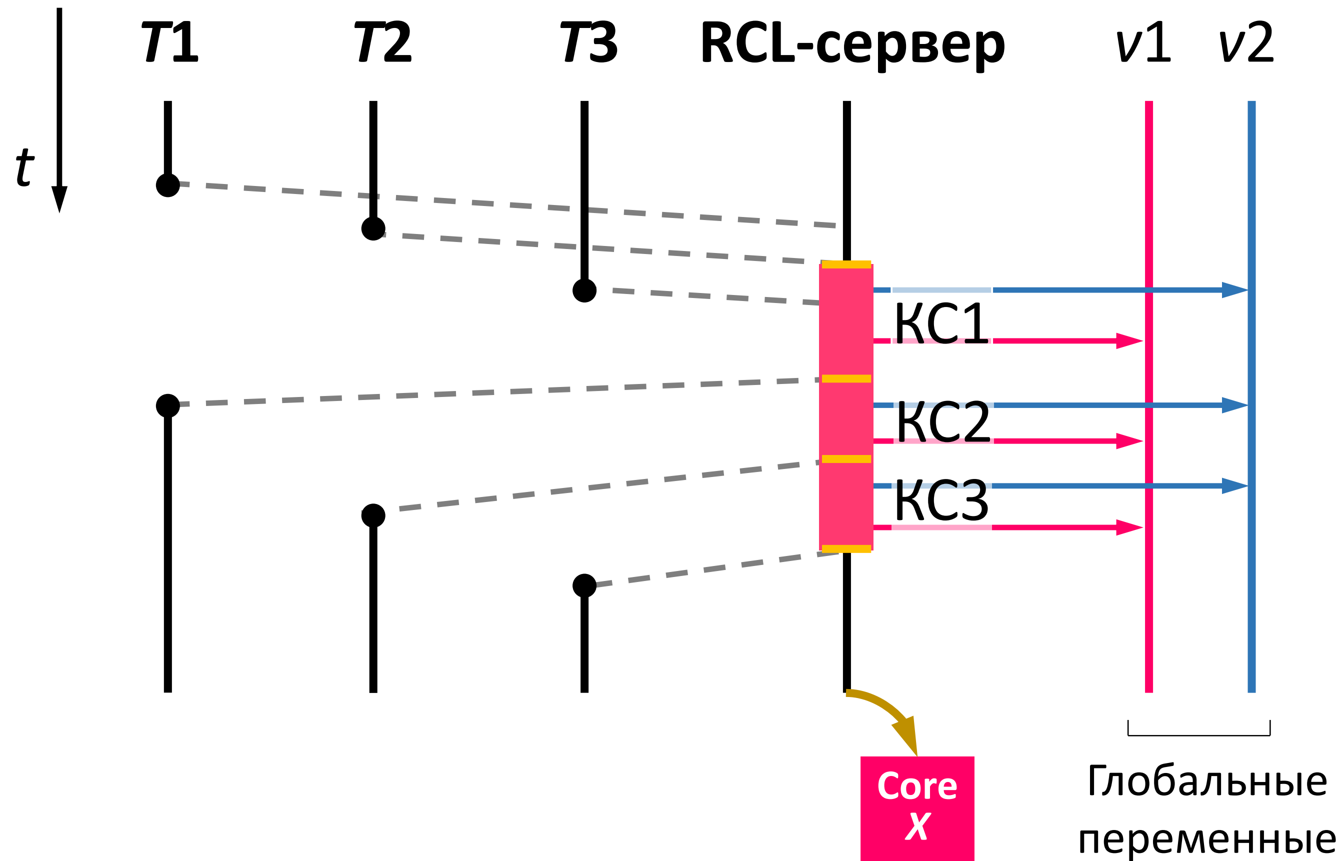
$\boldsymbol{t_3}$ – время реализации доступа к глобальным переменным

Локализация обращений к памяти в существующих алгоритмах блокировки:

- Spinlock ⇒ нет локализации
- PThread mutex ⇒ нет локализации
- MCS ⇒ нет локализации
- Flat combining ⇒ частичная локализация

Требуется разработка алгоритмов блокировки, обеспечивающих локализацию обращений к памяти.

Делегирование выполнения критических секций процессорным ядрам



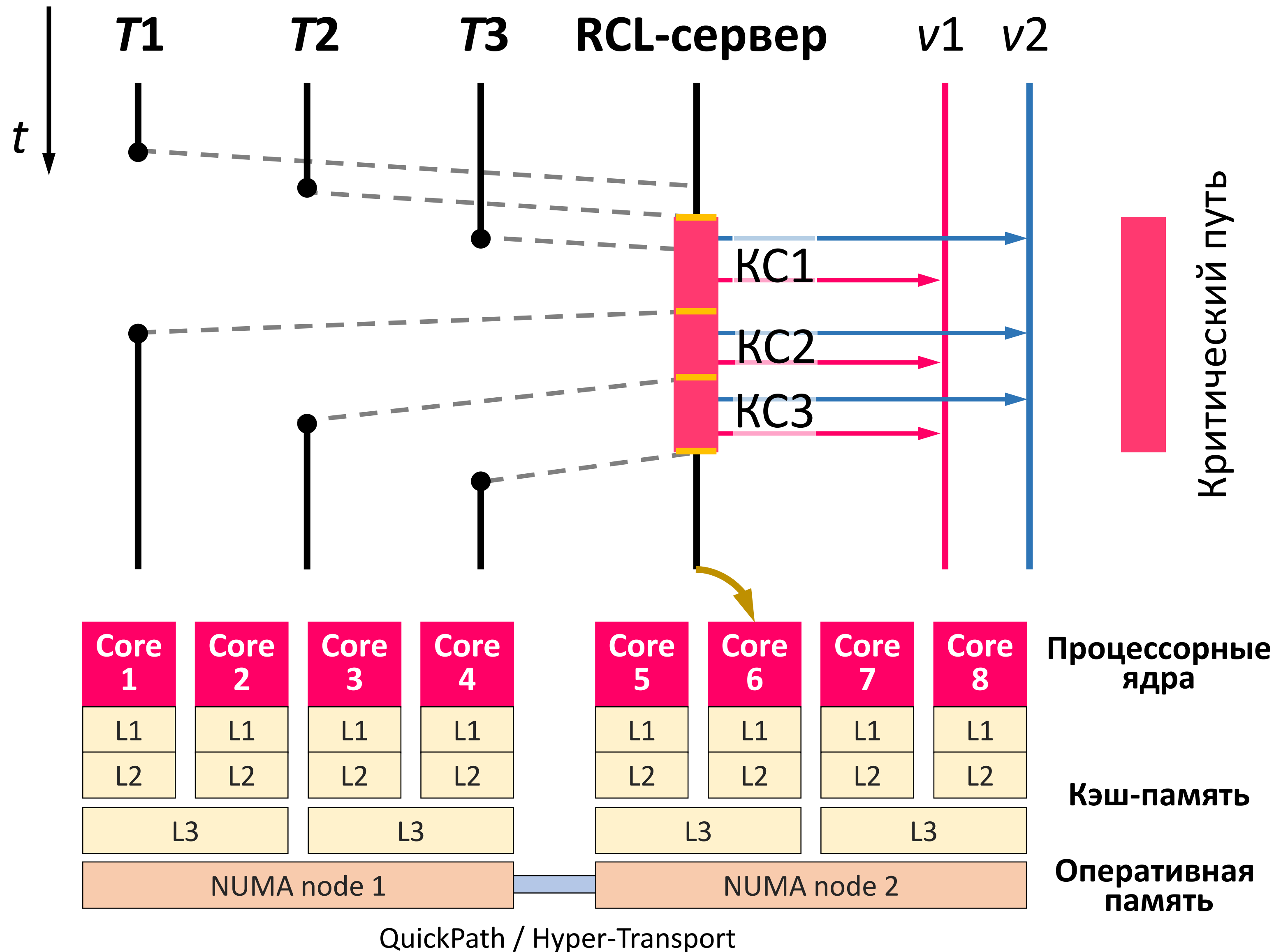
Метод делегирования выполнения критических секций выделенным процессорным ядрам (RCL) позволяет минимизировать критический путь выполнения критических секций

$$t = t_1 + t_2 + t_3$$

за счет минимизации

- времени t_2 передачи права выполнения критической секции
- и времени t_3 реализации доступа к глобальным переменным

Делегирование выполнения критических секций процессорным ядрам



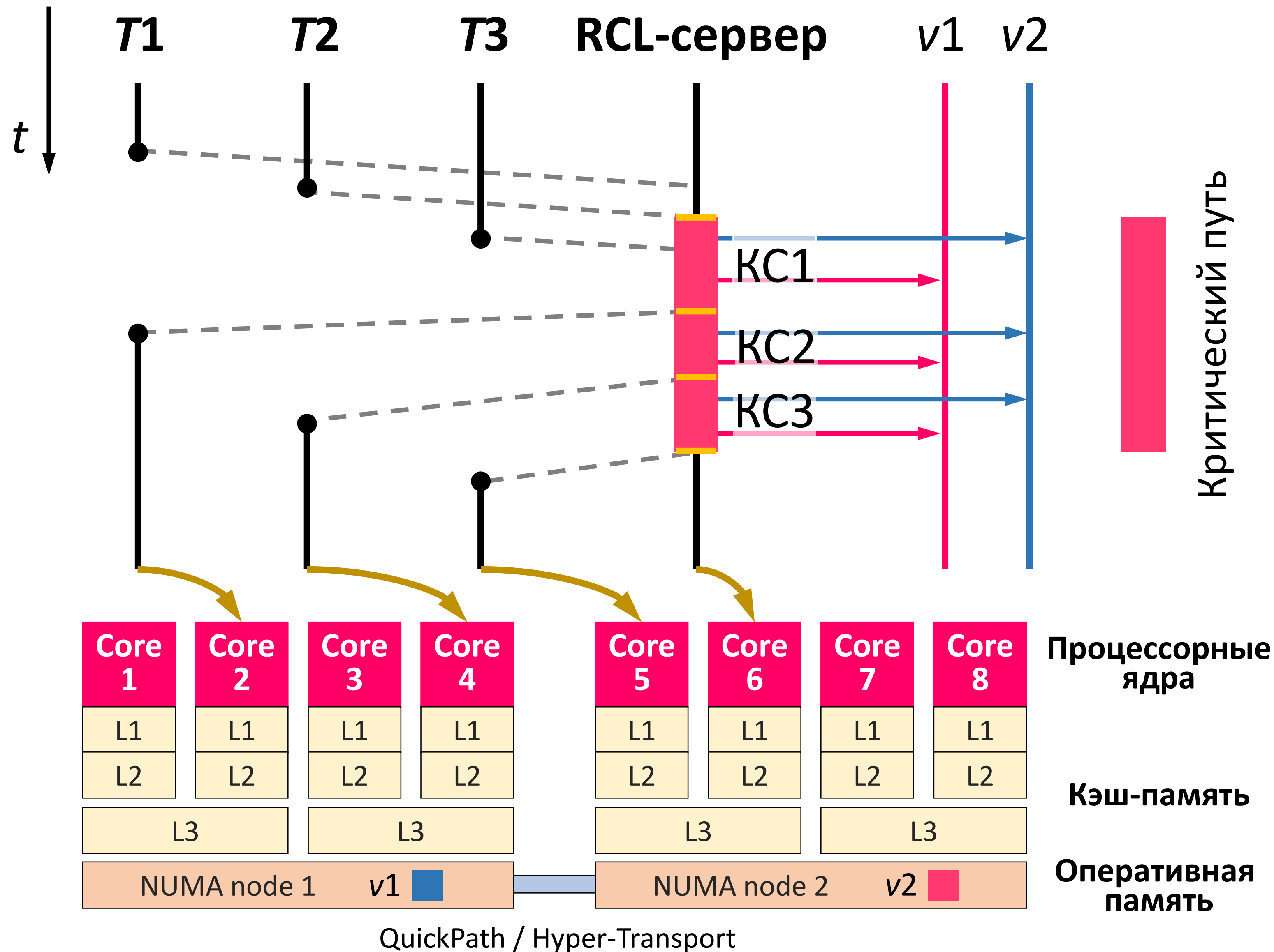
Метод делегирования выполнения критических секций выделенным процессорным ядрам (RCL) позволяет минимизировать критический путь выполнения критических секций

$$t = t_1 + t_2 + t_3$$

за счет минимизации

- времени t_2 передачи права выполнения критической секции
- и времени t_3 реализации доступа к глобальным переменным

Делегирование выполнения критических секций процессорным ядрам



Время выполнений критической секции (критический путь):

$$t = t_1 + t_2 + t_3$$

Время t_3 реализации доступа к глобальной переменной внутри критической секции зависит от

- того, на каком NUMA-узле выделена память для данной переменной
- на каком процессорном ядре функционирует RCL-сервер
- каким потоком и на каком процессорном ядре последний раз выполнялся доступ к данной переменной

Пример выполнения критической секции в RCL

```
liblock_lock_t lock;
const char* liblock_name = "rcl";

int global_var = 0;

void *cs(void* arg) {
 global_var++;
 return NULL;
}

void *thread(void* arg) {
 int i;
 for (i = 0; i < NITERS; i++) {
  liblock_exec(&lock, cs, NULL);
 }
 return NULL;
}
```

Выполнение критической секции

```
int main() {
 liblock_lock_init(liblock_name,
           &topology->hw_threads[0], &lock, 0);

 pthread_t tids[NTHREADS];
 for (int i = 0; i < NTHREADS; i++) {
  liblock_thread_create(&tids[i], NULL,
           thread, NULL);
 }

 for (int i = 0; i < NTHREADS; i++) {
  pthread_join(tids[i], NULL);
 }

 liblock_lock_destroy(&lock);
 return 0;
}
```

Инициализация блокировки

Создание потока

Пример выполнения критической секции в RCL

```
liblock_lock_t lock;
const char* liblock_name = "rcl";

int global_var = 0;

void *cs(void* arg) {
 global_var++;
 return NULL;
}

void *thread(void* arg) {
 int i;
 for (i = 0; i < NITERS; i++) {
  liblock_exec(&lock, cs, NULL);
 }
 return NULL;
}
```

Выполнение критической секции

```
int main() {
 liblock_lock_init(liblock_name,
          &topology->hw_threads[0], &lock, 0);
 pthread_t tids[NTHREADS];
 for (int i = 0; i < NTHREADS; i++) {
  liblock_thread_create(&tids[i], NULL,
             thread, NULL);
 }

 for (int i = 0; i < NTHREADS; i++) {
  pthread_join(tids[i], NULL);
 }

 liblock_lock_destroy(&lock);
 return 0;
}
```

Инициализация блокировки

1 Номер ядра

Создание потока

Пример выполнения критической секции в RCL

```
liblock_lock_t lock;
const char* liblock_name = "rcl";

int *global_var = NULL;

void *cs(void* arg) {
 (*global_var)++;
 return NULL;
}

void *thread(void* arg) {
 int i;
 for (i = 0; i < NITERS; i++) {
  liblock_exec(&lock, cs, NULL);
 }
 return NULL;
}
```

2 Выделение памяти в NUMA-системах

Выполнение критической секции

```
int main() {
 global_var = malloc(sizeof(*global_var));
 *global_var = 0;
 liblock_lock_init(liblock_name,
          &topology->hw_threads[0], &lock, 0);
 pthread_t tids[NTHREADS];
 for (int i = 0; i < NTHREADS; i++) {
  liblock_thread_create(&tids[i], NULL,
            thread, NULL);
 }
 for (int i = 0; i < NTHREADS; i++) {
  pthread_join(tids[i], NULL);
 }
 liblock_lock_destroy(&lock);
 return 0;
}
```

Инициализация блокировки

1 Номер ядра

Создание потока

Пример выполнения критической секции в RCL

```
liblock_lock_t lock;
const char* liblock_name = "rcl";

int *global_var = NULL;

void *cs(void* arg) {
  (*global_var)++;
  return NULL;
}

void *thread(void* arg) {
  int i;
  for (i = 0; i < NITERS; i++) {
    liblock_exec(&lock, cs, NULL);
  }
  return NULL;
}
```

```
int main() {
  global_var = malloc(sizeof(*global_var));
  *global_var = 0;
  liblock_lock_init(liblock_name,
            &topology->hw_threads[0], &lock, 0);

  pthread_t tids[NTHREADS];
  for (int i = 0; i < NTHREADS; i++) {
    liblock_thread_create_and_bind(
      &topology->hw_threads[1],
      0, &tids[i], NULL, thread, NULL);
  }
  for (int i = 0; i < NTHREADS; i++) {
    pthread_join(tids[i], NULL);
  }
  liblock_lock_destroy(&lock);
```

1 Номер ядра

2 Выделение памяти в NUMA-системах

3 **Пользователь** должен самостоятельно учитывать структуру ВС

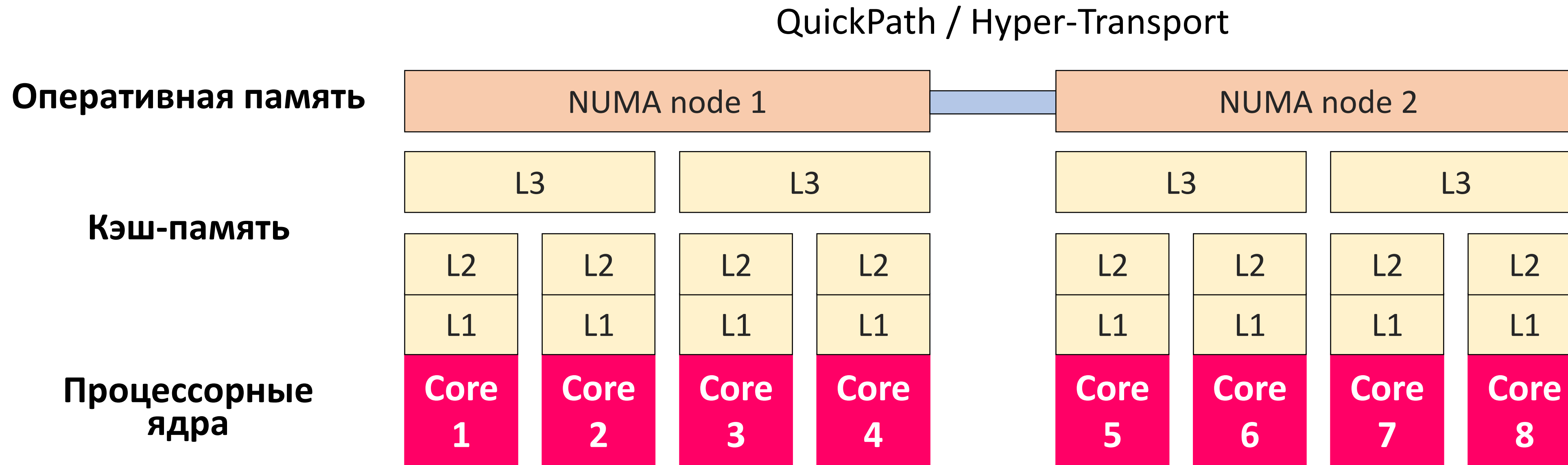
Инициализация блокировки

Выполнение критической секции

Недостатки текущей реализации RCL

1. **Отсутствие механизма автоматического выбора процессорного ядра при инициализации RCL-блокировок.**
   Пользователь должен самостоятельно выбирать процессорные ядра для инициализации RCL-серверов, учитывая при этом иерархическую структуру распределённой ВС.
2. **Выделение памяти без учёта неоднородного доступа к памяти.**
   Время работы многопоточной программы в NUMA-системах с использованием RCL существенно зависит от того, на каком узле функционирует RCL-сервер и с какого NUMA-узла выполняется выделение памяти.
3. **Нет возможности автоматической привязки рабочих потоков с учётом привязки RCL-серверов и иерархической структуры ВС.**
   Программист при создании потоков должен самостоятельно учитывать расположение RCL-серверов в ВС, привязку уже созданных потоков, а также иерархическую структуру ВС.

Модель многопроцессорной вычислительной системы



$P = \{1,2,\dots,N\}$ – множество процессорных ядер распределённой ВС
L – количество иерархических уровней системы
n_l – количество элементов на уровне l
n_{lk} – количество прямых дочерних узлов элемента $k \in \{1,2,\dots,nl\}$ на уровне l
c_{lk} – количество процессорных ядер, принадлежащих потомкам элемента k на уровне l
C_{lk} – множество процессорных ядер, принадлежащих потомкам элемента k на уровне l

Khoroshevsky V., Kurnosov M. **Mapping Parallel Programs into Hierarchical Distributed Computer Systems** // Proceedings of 4th International Conference “Software and Data Technologies (ICSOFT 2009)”. - Sofia: INSTICC, 2009. - Vol. 2. - P. 123-128.

Алгоритм RCLLockInitNUMA инициализации RCL-блокировки

```
    INITLIBRARY()
1   TRYSETMEMBIND(defaultNode)
2   topology = INITHWLOCTOPOLOGY()
```

Алгоритм RCLLockInitNUMA инициализации RCL-блокировки

```
    INITLIBRARY()
1   TRYSETMEMBIND(defaultNode)
2   topology = INITHWLOCTOPOLOGY()
```

```
    RCLLOCKINITNUMA()
1   node_usage = GETNODESUSAGE(node_usage)
2   TRYSETMEMBIND(nodes_usage)
3   core = GETFREECORE(nodes_usage)
4   RCLLOCKINITDEFAULT(core)
```

Алгоритм RCLLockInitNUMA инициализации RCL-блокировки

```
    INITLIBRARY()
1   TRYSETMEMBIND(defaultNode)
2   topology = INITHWLOCTOPOLOGY()
```

```
    RCLLOCKINITNUMA()
1   node_usage = GETNODESUSAGE(node_usage)
2   TRYSETMEMBIND(nodes_usage)
3   core = GETFREECORE(nodes_usage)
4   RCLLOCKINITDEFAULT(core)
```

```
    GETNODESUSAGE(node_usage)
1   for core = 1 to N do
2     if ISSERVERRUNNING(core) then
3       nb_free_cores = nb_free_cores + 1
4     else
5       nodes_usage[GETNODE(core)]++
```

Алгоритм RCLLockInitNUMA инициализации RCL-блокировки

```
    InitLibrary()
1   TrySetMembind(defaultNode)
2   topology = InitHwlocTopology()
```

```
    RCLLockInitNUMA()
1   node_usage = GetNodesUsage(node_usage)
2   TrySetMembind(nodes_usage)
3   core = GetFreeCore(nodes_usage)
4   RCLLockInitDefault(core)
```

```
    GetNodesUsage(node_usage)
```

```
    TrySetMembind(nodes_usage)
1   for i = 0 to nnodes do
2     if nodes_usage[i] > 0 then
3       nb_busy_nodes++
4       node = i
5   if nb_busy_nodes = 1 then
6     SetMemBind(node)
```

Алгоритм RCLLockInitNUMA инициализации RCL-блокировки

```
    InitLibrary()
1   TrySetMemBind(defaultNode)
2   topology = InitHwlocTopology()
```

```
    RCLLockInitNUMA()
1   node_usage = GetNodesUsage(node_usage)
2   TrySetMemBind(nodes_usage)
3   core = GetFreeCore(nodes_usage)
4   RCLLockInitDefault(core)
```

```
    GetNodesUsage(node_usage)
```

```
    TrySetMemBind(nodes_usage)
```

```
    GetFreeCore(nodes_usage)
1   if nb_free_cores ≤ 1 then
2     core = GetNextCoreInRRFashion()
3   else
4     node = GetMostBusyNode(nodes_usage)
5     for each core in node do
6       if !IsServerRunning(core) then
7         return core
```

Алгоритм RCLHierarchicalAffinity привязки потоков

```
    RCLHierarchicalAffinity(thread_attr)
1   if IsRegularThread(thread_attr) then
2     for cpu = 1 to N do
3       if IsRCLOnCPU(cpu) then
4         nthreads_per_cpu = 0
5         obj = cpu
6         do
7           if NoCoverObj(obj) then
8             nthr_per_cpu++
9             obj = cpu
10          obj = GetCoverObject(obj)
11          free_cpu =
12            GetFreeCPU(obj, nthr_per_cpu);
13        while free_cpu = Ø
14        SetAffinity(free_cpu, thread_attr)
```

Алгоритм RCLHierarchicalAffinity привязки потоков

```
     RCLHIERARCHICALAFFINITY(thread_attr)
1    if ISREGULARTHREAD(thread_attr) then
2      for cpu = 1 to N do
3        if ISRCLONCPU(cpu) then
4          nthreads_per_cpu = 0
5          obj = cpu
6          do
7            if NOCOVEROBJ(obj) then
8              nthr_per_cpu++
9              obj = cpu
10           obj = GETCOVEROBJECT(obj)
11           free_cpu =
12             GETFREECPU(obj, nthr_per_cpu);
13         while free_cpu = Ø
14         SETAFFINITY(free_cpu, thread_attr)
```

```
     GETCOVEROBJECT(obj)
1    ncpus = GetCPUCountInsideObj(obj)
2    do
3      ncpus_prev = ncpus
4      obj = GetParent(obj)
5      ncpus = GetCPUCountInsideObj(obj)
6      GetCPUCountInsideObj(obj)
7    while ncpus = ncpus_prev
8    return obj
```

Алгоритм RCLHierarchicalAffinity привязки потоков

```
    RCLHierarchicalAffinity(thread_attr)
1   if IsRegularThread(thread_attr) then
2     for cpu = 1 to N do
3       if IsRCLOnCPU(cpu) then
4         nthreads_per_cpu = 0
5         obj = cpu
6         do
7           if NoCoverObj(obj) then
8             nthr_per_cpu++
9             obj = cpu
10          obj = GetCoverObject(obj)
11          free_cpu =
12            GetFreeCPU(obj, nthr_per_cpu);
13        while free_cpu = Ø
14        SetAffinity(free_cpu, thread_attr)
```

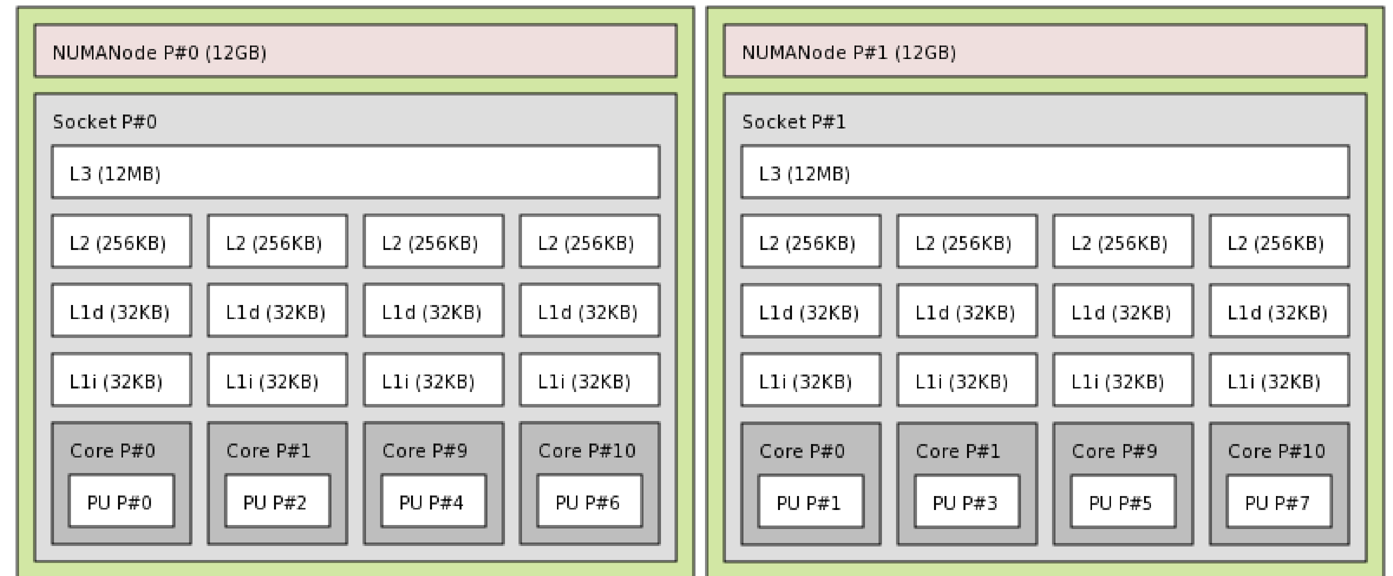
```
    GetCoverObject(node_usage)
```

```
    GetFreeCPU(obj, nthr_per_cpu)
1   for core in obj do
2     if CPUIsFree(core) and
3        busy_cpus[core] ≤ nthr_per_cpu then
4       return core
```

Организация экспериментов – аппаратное и программное обеспечение

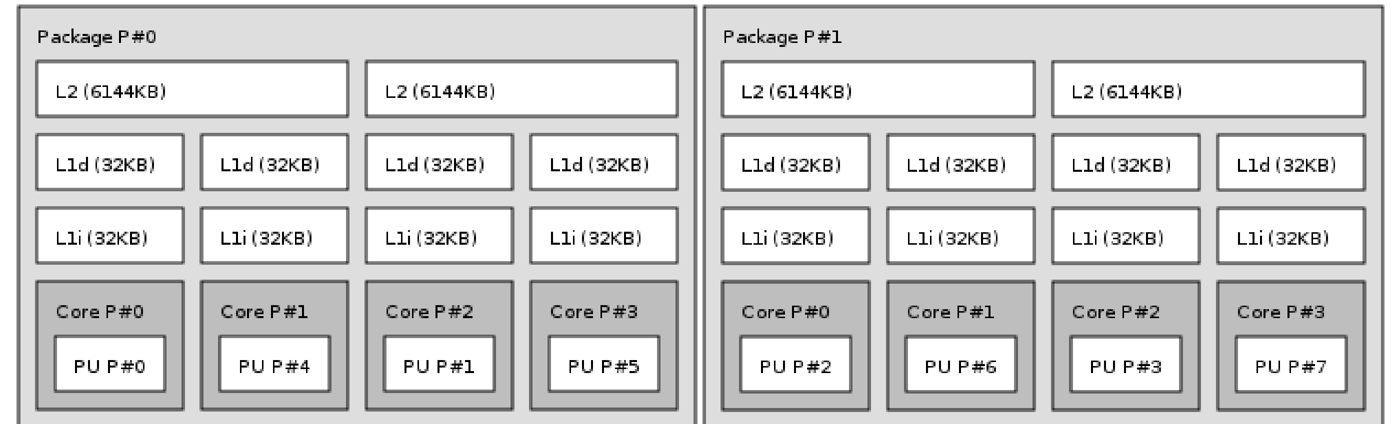
Узел кластера Oak (NUMA-система)
ЦПВТ СибГУТИ

- 2 x Intel Xeon E5620
  (2.4 GHz, 4 ядер) – 8 ядер
- Кэш: L1 (32 KB), L2 (256 KB), L3 (12 MB)
- Оперативная память 24 GB
- Соотношение скорости доступа к NUMA-сегментам памяти: 10:21.



Узел кластера Jet (SMP-система)
ЦПВТ СибГУТИ

- 2 x Intel Xeon E5420
  (2.4 GHz, 4 ядер) – 8 ядер
- Кэш: L1 (32 KB), L2 (6 MB)
- Оперативная память 8 GB



Программное обеспечение: GNU/Linux Fedora 20 (Jet), CentOS 6.0 (Oak), компилятор GCC 5.3.0

Организация экспериментов – тестовые программы

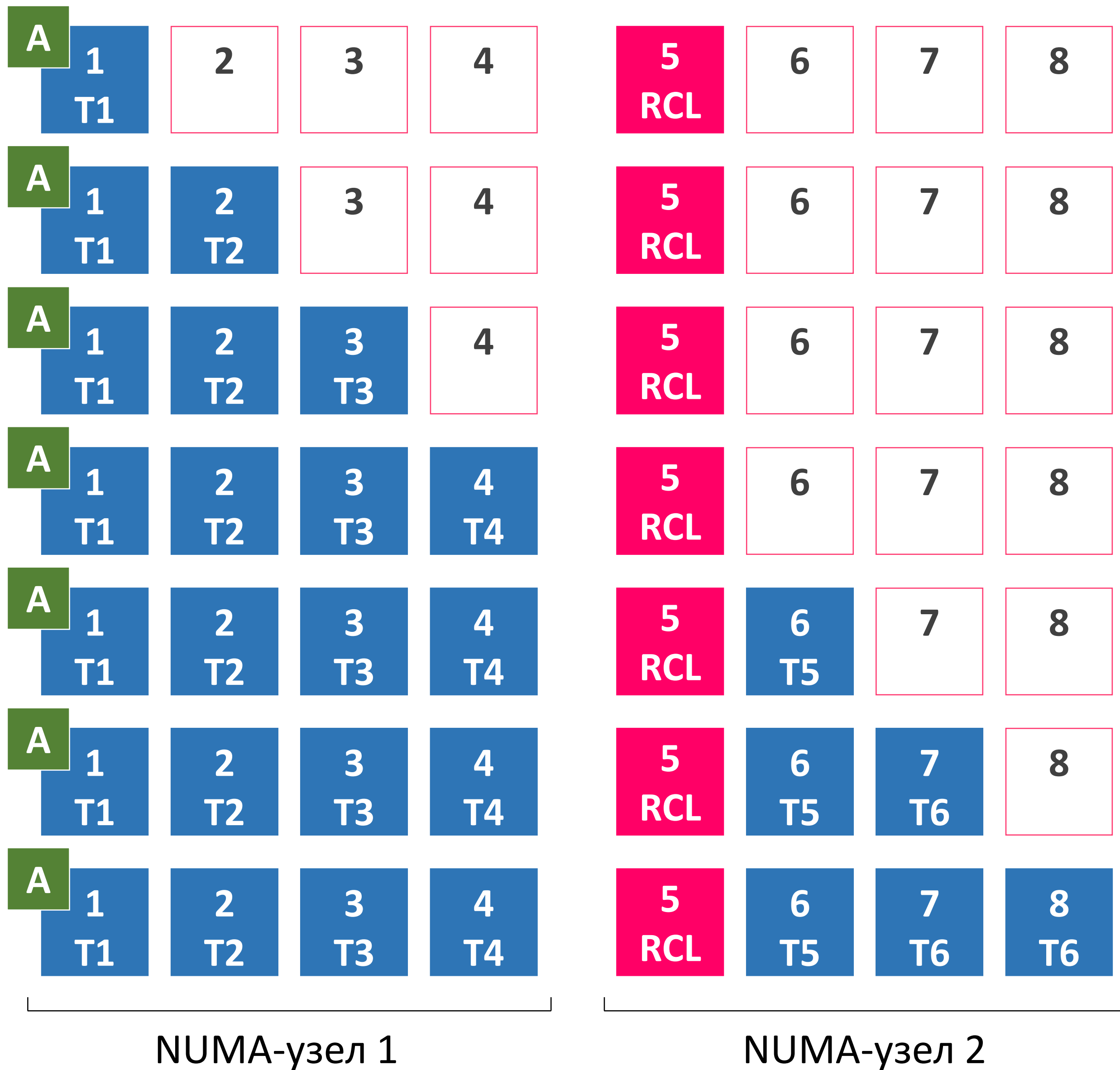
Синтетический тест

- Циклический доступ к элементам целочисленного массива
- Размер массива $b = 5 \times 10^8$ элементов
- Количество операций $n = 10^8/p$
- Шаблон операции: увеличение переменной на 1
- Шаблоны доступа к элементам
  - Последовательный доступ (sequential access)
  - Случайный доступ (random access)
  - Доступ с интервалом $s = 20$ элементов (strided access)

Показатель эффективности

Пропускная способность: $b = n/t$, где t – время выполнения теста.

Привязка потоков к процессорным ядрам



1 рабочий поток

2 рабочих потока

3 рабочих потока

4 рабочих потока

5 рабочих потоков

6 рабочих потоков

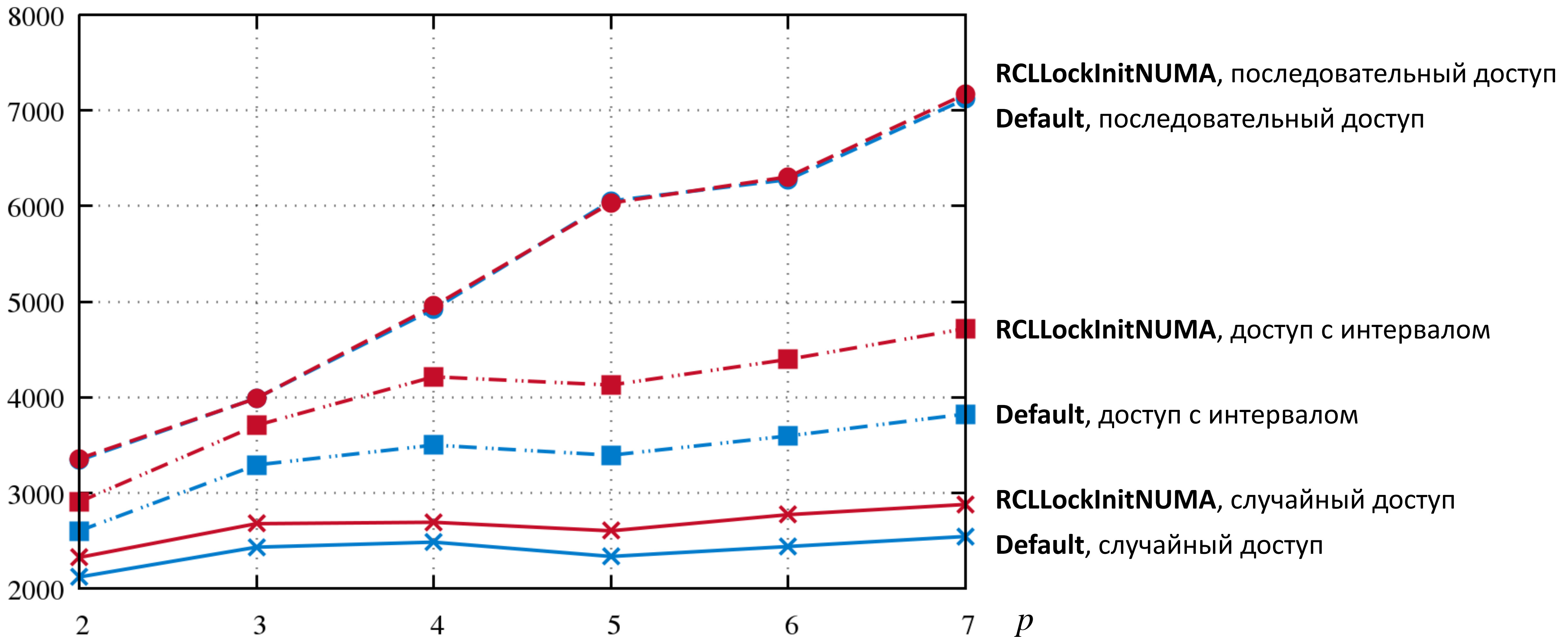
7 рабочих потоков

1 T1 – рабочий поток

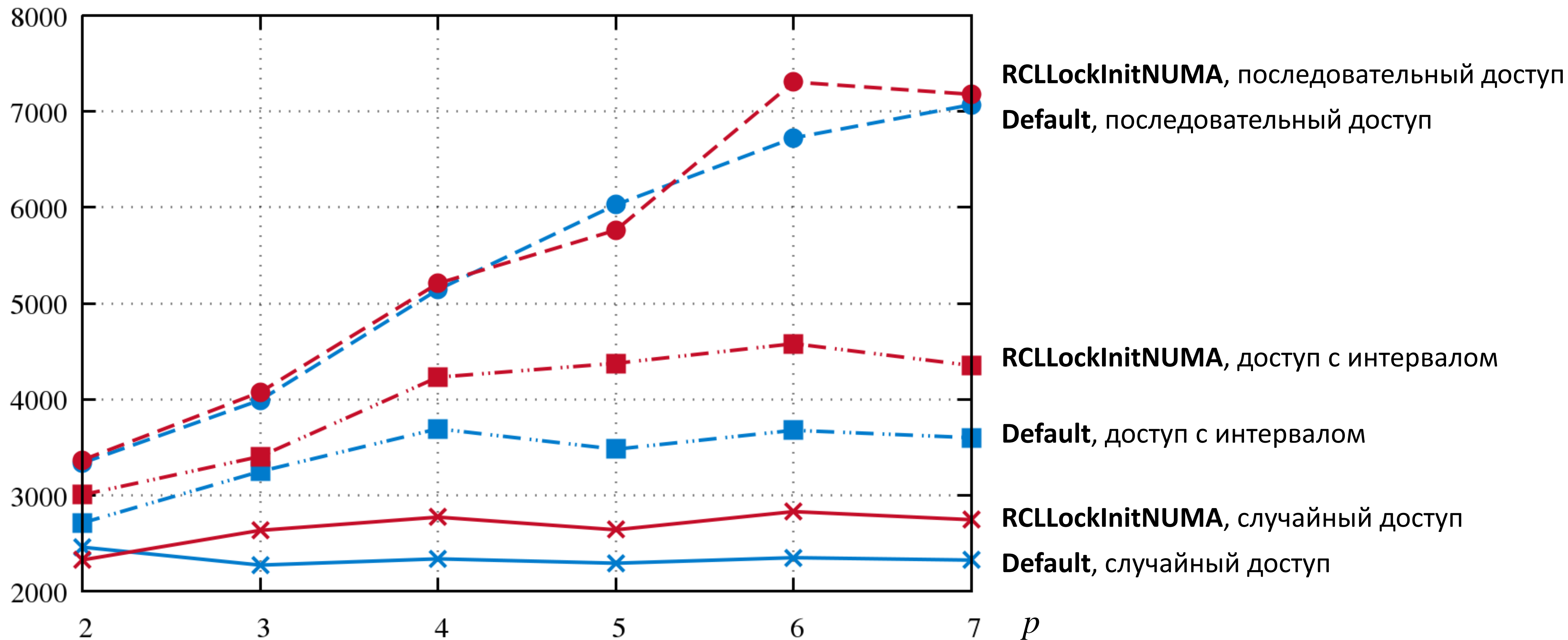
5 RCL – RCL-сервер

A – поток, выполняющий аллоцирование памяти

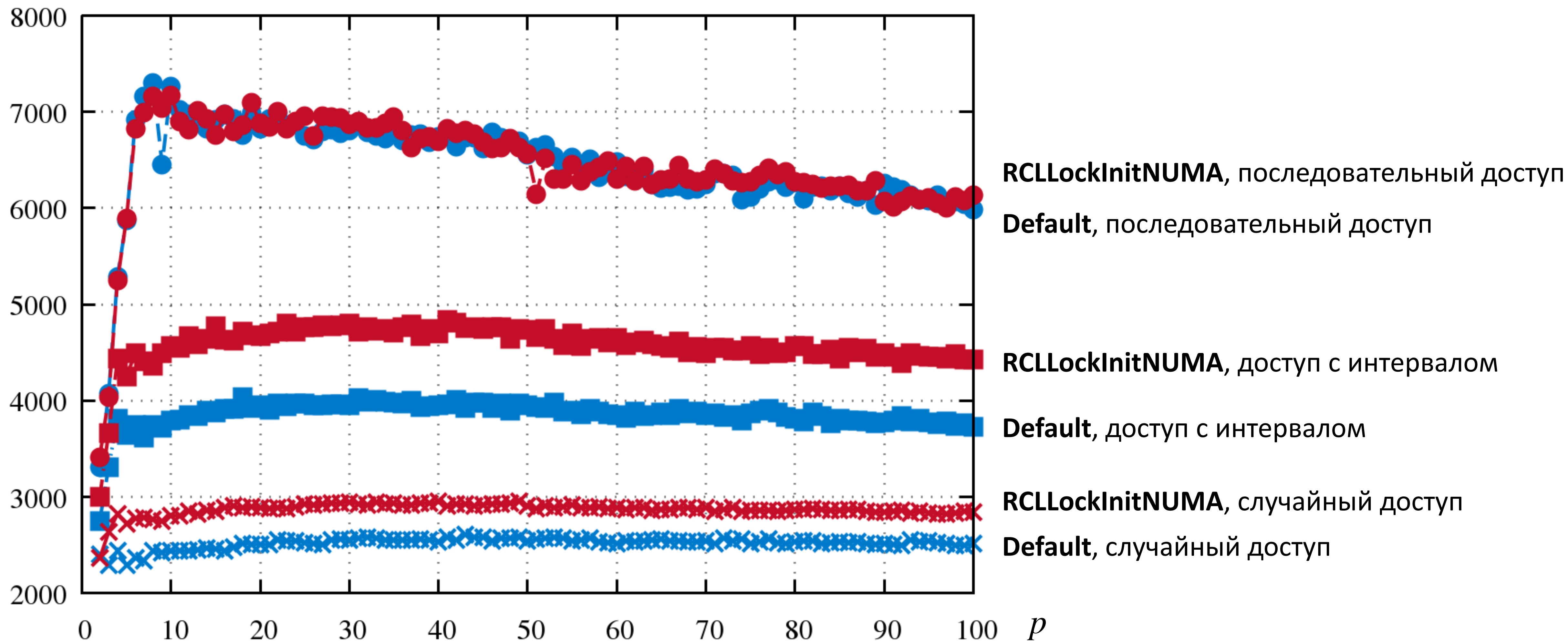
Результаты экспериментов, алгоритм RCLLockInitNUMA



Результаты экспериментов, алгоритм RCLLockInitNUMA



Результаты экспериментов, алгоритм RCLLockInitNUMA



Результаты экспериментов, алгоритм RCLHierarchicalAffinity

b, 1000 опер/с

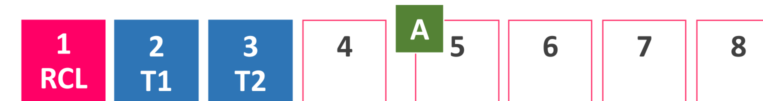
Кластер Oak, 2 потока

	Default, случайный доступ	Default, последовательный доступ	Default, доступ с интервалом	RCLLockInitNUMA, случайный доступ	RCLLockInitNUMA, последовательный доступ	RCLLockInitNUMA, доступ с интервалом
Привязка 1						
Привязка 2						
Привязка 3						

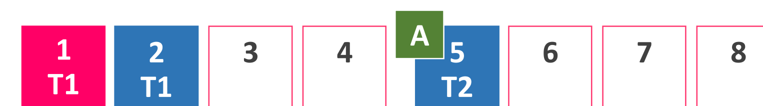
Шкала: 2 000, 2 500, 3 000, 3 500, 4 000, 4 500, 5 000, 5 500, 6 000, 6 500, 7 000, 7 500, 8 000

RCLHierarchicalAffinity:

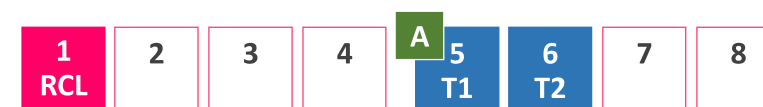
Привязка 1: (2, 3)



Привязка 2: (2, 5)



Привязка 3: (5, 6)



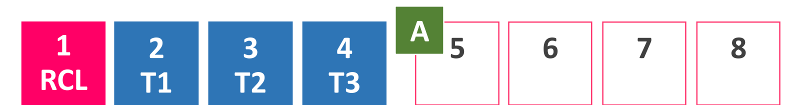
Результаты экспериментов, алгоритм RCLHierarchicalAffinity

b, 1000 опер/с

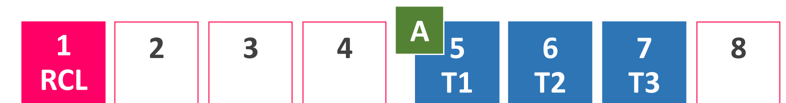
Кластер Oak, 3 потока

RCLHierarchicalAffinity:

Привязка 1: (2, 3, 4)



Привязка 2: (5, 6, 7)



Привязка 3: (2, 3, 5)

Привязка 4: (2, 5, 6)

Результаты экспериментов, алгоритм RCLHierarchicalAffinity

b, 1000 опер/с

Кластер Oak, 4 потока

Y-axis: 2 000, 2 500, 3 000, 3 500, 4 000, 4 500, 5 000, 5 500, 6 000

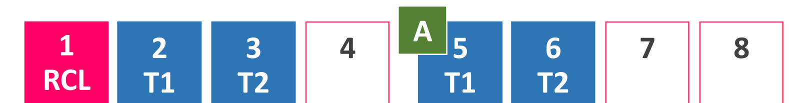
X-axis: Привязка 1, Привязка 2, Привязка 3, Привязка 4

Legend:
- Default, случайный доступ
- Default, последовательный доступ
- Default, доступ с интервалом
- RCLLockInitNUMA, случайный доступ
- RCLLockInitNUMA, последовательный доступ
- RCLLockInitNUMA, доступ с интервалом

RCLHierarchicalAffinity:

Привязка 1: (2, 3, 4, 5)

Привязка 2: (2, 3, 5, 6)



Привязка 3: (2, 5, 6, 7)

Привязка 4: (5, 6, 7, 8)

Результаты экспериментов, алгоритм RCLHierarchicalAffinity

b, 1000 опер/с

Кластер Oak, 5 потоков

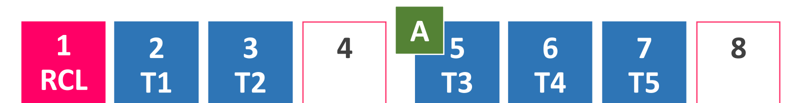
	Default, случайный доступ	Default, последовательный доступ	Default, доступ с интервалом	RCLLockInitNUMA, случайный доступ	RCLLockInitNUMA, последовательный доступ	RCLLockInitNUMA, доступ с интервалом
Привязка 1						
Привязка 2						
Привязка 3						

Ось: 2 000, 2 500, 3 000, 3 500, 4 000, 4 500, 5 000, 5 500, 6 000, 6 500, 7 000

RCLHierarchicalAffinity:

Привязка 1: (2, 3, 4, 5)

Привязка 2: (2, 3, 5, 6)



Привязка 3: (2, 3, 5)

Результаты экспериментов, алгоритм RCLHierarchicalAffinity

b, 1000 опер/с

Кластер Jet, 2 потока

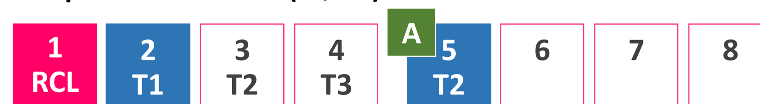
3 600
3 400
3 200
3 000
2 800
2 600
2 400
2 200
2 000

Привязка 1 Привязка 2 Привязка 3 Привязка 4 Привязка 5

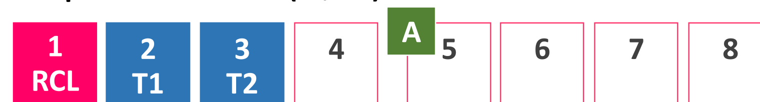
Default, случайный доступ
Default, доступ с интервалом
Default, последовательный доступ

RCLHierarchicalAffinity:

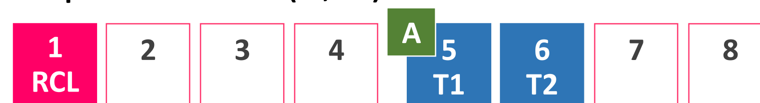
Привязка 1: (2, 5)



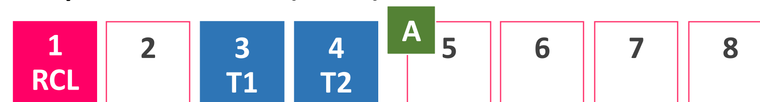
Привязка 2: (2, 3)



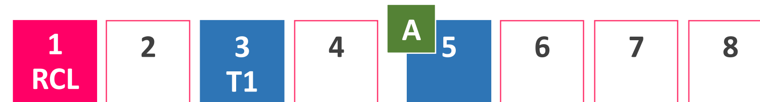
Привязка 3: (5, 6)



Привязка 4: (3, 4)



Привязка 5: (3, 5)



Результаты экспериментов, алгоритм RCLHierarchicalAffinity

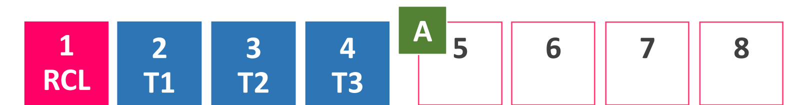
b, 1000 опер/с

	Default, случайный доступ	Default, последовательный доступ	Default, доступ с интервалом

Кластер Jet, 3 потока

RCLHierarchicalAffinity:

Привязка 1: (2, 3, 4)



Привязка 2: (5, 6, 7)

Привязка 3: (2, 3, 5)

Привязка 4: (2, 5, 6)

Результаты экспериментов, алгоритм RCLHierarchicalAffinity

b, 1000 опер/с

	Default, случайный доступ	Default, последовательный доступ	Default, доступ с интервалом
Привязка 1			
Привязка 2			
Привязка 3			
Привязка 4			

Ось: 2 400, 2 600, 2 800, 3 000, 3 200, 3 400, 3 600, 3 800, 4 000, 4 200, 4 400, 4 600, 4 800, 5 000

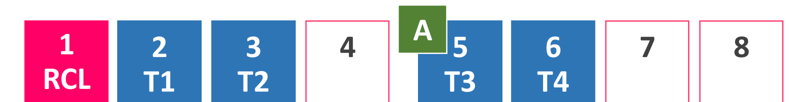
Default, случайный доступ Default, доступ с интервалом
Default, последовательный доступ

Кластер Jet, 4 потока

RCLHierarchicalAffinity:

Привязка 1: (2, 3, 4, 5)

Привязка 2: (2, 3, 5, 6)



Привязка 3: (2, 5, 6, 7)

Привязка 4: (5, 6, 7, 8)

Результаты экспериментов, алгоритм RCLHierarchicalAffinity

b, 1000 опер/с

Кластер Jet, 5 потоков

	Default, случайный доступ	Default, последовательный доступ	Default, доступ с интервалом

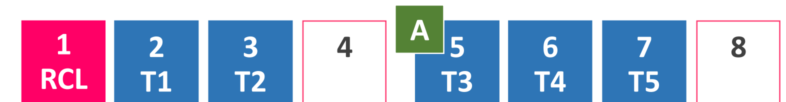
Ось: 2 400; 2 600; 2 800; 3 000; 3 200; 3 400; 3 600; 3 800; 4 000; 4 200; 4 400; 4 600; 4 800; 5 000

Привязка 1 | Привязка 2 | Привязка 3

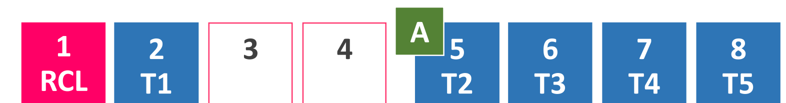
RCLHierarchicalAffinity:

Привязка 1: (2, 3, 4, 5)

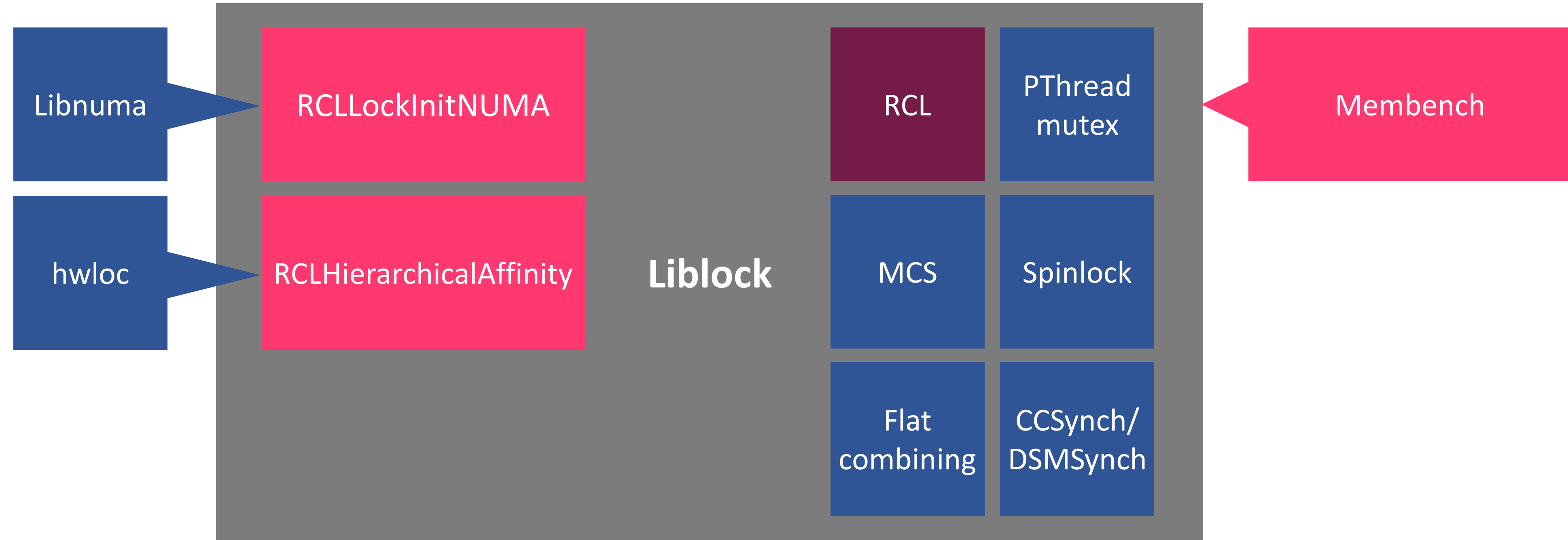
Привязка 2: (2, 3, 5, 6)



Привязка 3: (2, 3, 5)



Программная реализация алгоритмов



- **Liblock** – библиотека, реализующая алгоритмы блокировки
- Алгоритмы RCLLockInitNUMA и RCLHierarchicalAffinity реализованы в виде функций библиотеки Liblock
- **Libnuma** – библиотека, реализующая привязку выделения памяти к NUMA-узлам
- **hwloc** – получение информации об иерархической структуре ВС
- **Membench** – синтетический тест

Спасибо за внимание!

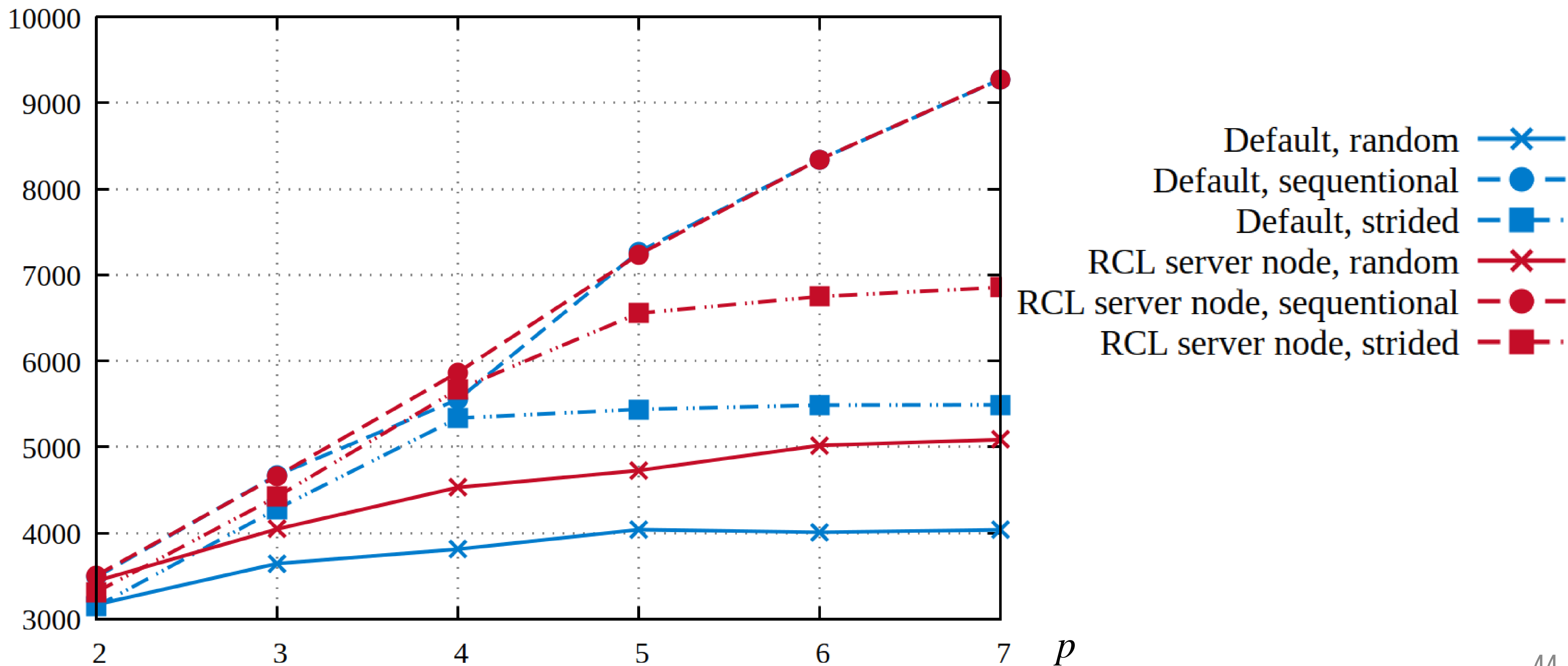
Таблица запросов в RCL

Пример выполнения критической секции в RCL

```
liblock_lock_t lock;
const char* liblock_name = "rcl";

int *global_var = NULL;

void *cs(void* arg) {
  (*global_var)++;
  return NULL;
}

void *thread(void* arg) {
  int i;
  for (i = 0; i < NITERS; i++) {
   liblock_exec(&lock, cs, NULL);
  }
  return NULL;
}
```

```
int main() {
  global_var = malloc(sizeof(*global_var));
  *global_var = 0;
  liblock_lock_init(liblock_name,
                    &topology->hw_threads[0],

  pthread_t tids[NTHREADS];
  for (int i = 0; i < NTHREADS; i++) {
   liblock_thread_create(&tids[i], NULL,
                         thread, NULL);
  }
  for (int i = 0; i < NTHREADS; i++) {
   pthread_join(tids[i], NULL);
  }
  liblock_lock_destroy(&lock);
  return 0;
}
```

Результаты экспериментов



Алгоритм RCLlockInitNUMA иницилазиации блокировки

```
    InitLibrary()
1   TrySetAffinity(defaultNode)
2   topology = InitHwlocTopology()
```

```
    RCLLockInitNUMA()
1   node_usage = GetNodesUsage(node_usage)
2   TrySetMembind(nodes_usage)
3   core = GetFreeCore(nodes_usage)
4   RCLLockInitDefault(core)
```

```
    GetNodesUsage(node_usage)
1   for core = 1 to N do
2     if IsServerRunning(core) then
3       nb_free_cores = nb_free_cores + 1
4     else
5       nodes_usage[GetNode(core)]++
```

```
    GetFreeCore(nodes_usage)
1   if nb_free_cores ≤ 1 then
2     core = GetNextCoreInRRFashion()
3   else
4     node = GetMostBusyNode(nodes_usage)
5     for each core in node do
6       if !IsServerRunning(core) then
7         return core
```

Организация экспериментов – аппаратное и программное обеспечение

Конфигурация подсистемы

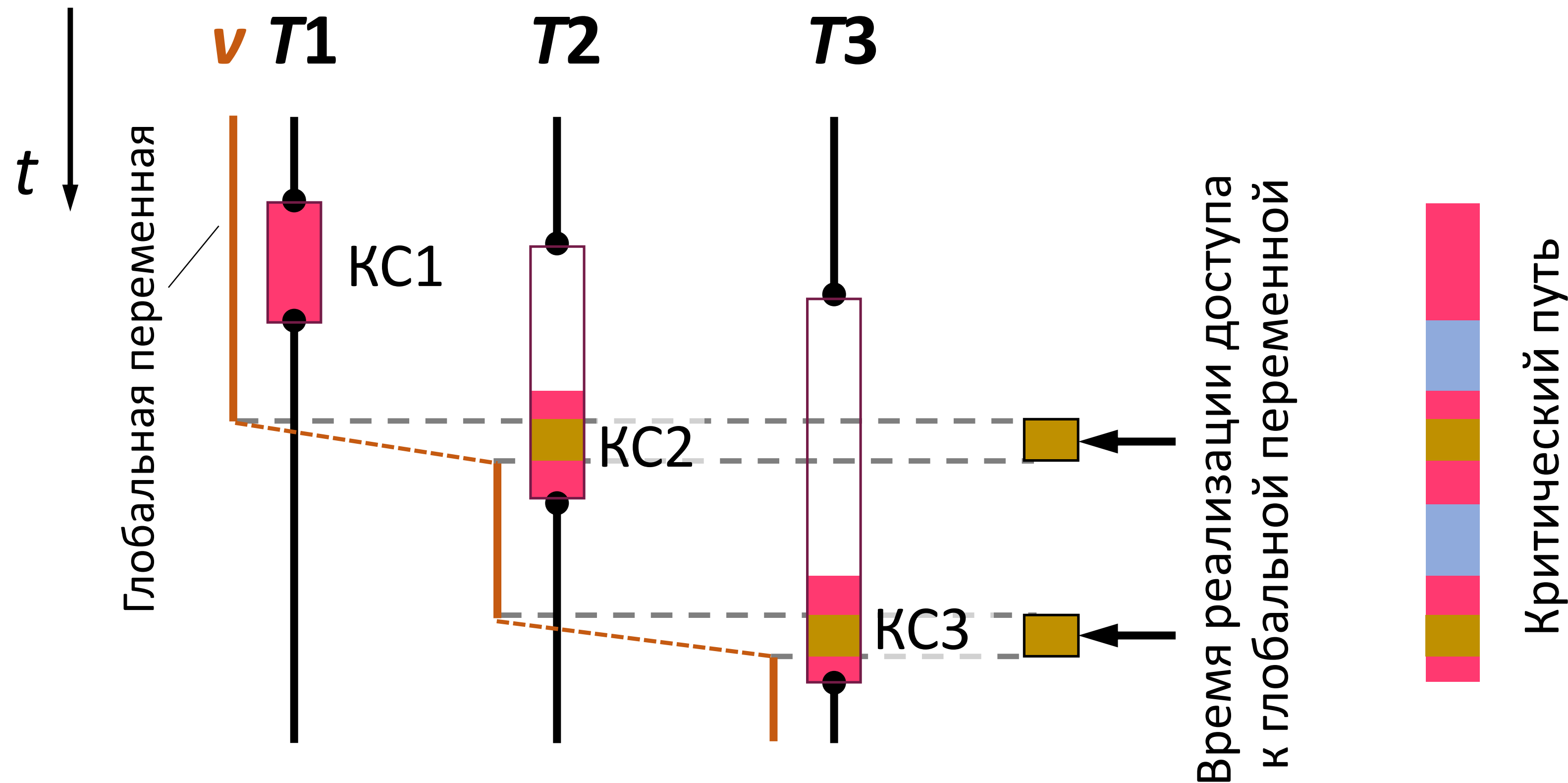
- Узел кластера Oak (NUMA-система)
  - 2 x Intel Xeon E5620 (2.4 GHz, 4 ядер)
  - Кэш L1 (32 KB), L2 (256 KB), L3 (12 MB)
  - Оперативная память 24 GB

    Соотношение скорости доступа к локальному и удалённому сегментам памяти: 10:21.
- Узел кластера Jet (SMP-система)
  - 2 x Intel Xeon E5420 (2.4 GHz, 4 ядер)
  - Кэш: L1 (32 KB), L2 (6 MB)
  - Оперативная память 8 GB

Программное обеспечение

- GNU/Linux Fedora 20 (Jet), CentOS 6.0 (Oak)
- GCC 5.3.0

Делегирование выполнения критических секций процессорным ядрам



Время выполнений критической секции (критический путь):

$$t = t_1 + t_2 + t_3,$$

где

t_1 – время выполнения инструкций критической секции,

t_2 – время передачи права выполнения критической секции,

t_3 – время реализации доступа к глобальным переменным

Требуется разработка алгоритмов блокировки, обеспечивающих локализацию обращений к памяти.

Организация экспериментов – тестовые программы

Синтетический тест

- Циклический доступ к элементам целочисленного массива
- Размер массива $b = 5 \times 108$ элементов
- Количество операций $n = 108/p$
- Шаблон операции: увеличение переменной на 1
- Шаблоны доступа к элементам
  - Последовательный доступ (sequential access)
  - Случайный доступ (random access)
  - Доступ с интервалом $s = 1000$ элементов (strided access)

Реальные многопоточные программы

- **SPLASH2** – пакет отраслевых тестов и численных методов (kernels)
  - **Barnes, FMM** – моделирование взаимодействия физические тел (метод N тел)
  - **Cholesky, LU** – методы линейной алгебры решения СЛАУ
  - **FFT** – быстрое преобразование Фурье
  - **Ocean** – моделирование течений в мировом океане
  - **Radiosity, Raytrace, Volrend** – моделирование сцен трёхмерной графики
  - **Radix** – поразрядная сортировка
  - **Water-Nsquared, Water-Spatial** – задачи вычислительной гидродинамики

Организация экспериментов – тестовые программы

Реальные многопоточные программы

- **Phoenix** – реализация метода MapReduce для многоядерных ВС
  - **Histogram** – определение гистограммы RGB-изображения
  - **Linear regression** – решение задачи аппроксимации по множеству точек
  - **String match** – поиск строки в текстовом файле
  - **Word count** – определение частоты встречаемости слов в документе
  - **Matrix multiply** – умножение целочисленных матриц
  - **Kmeans** – алгоритм кластеризации множества точек
  - **PCA** – метод главных компонент
  - **Reverse index** – составление словаря ссылок по заданному HTML-файлу