Курс «Архитектурно-ориентированная оптимизация кода»

Коллективные операции MPI

Михаил Курносов

MPI Communication Routines

- **Type of collective communication operations**
  - **One-to-all**: Broadcast, Scatter
  - **All-to-one**: Gather, Reduce, Scan
  - **All-to-all**: Allgather, Alltoall, Allreduce, Reduce_scatter, Reduce_scatter_block
  - **Barrier**

- **MPI Collective Operations** (MPI 4.0)
  - **Blocking**: MPI_Bcast(buf, count, dtype, root)
  - **Nonblocking**: MPI_Ibcast(buf, count, dtype, root, req) + MPI_Wait()
  - **Persistent**: MPI_Bcast_init(buf, count, dtype, root, req) + MPI_Start()
  - **Neighborhood Collective Communication**: Gather, Alltoall

MPI Collective Communication Algorithms

Flat point-to-point based (send/recv)

Only number of processes and message size are known

- **One-to-all (Broadcast, Scatter)**
  - Binary tree, Binomial tree, *k*-nomial tree $O(\log(p))$
  - Flat tree $O(p)$, *k*-chain $O(p)$, pipeline
  - Scatter binomial tree + allgather recursive doubling
- **All-to-one (Gather, Reduce, Scan)**
  - Binomial tree, *k*-nomial tree, binary tree,
  - *k*-chain, pipeline, linear
  - Rabesifner's algorithm
- **All-to-all (Allgather, Alltoall, Allreduce, Reduce_scatter, Reduce_scatter_block)**
  - Bruck, recursive doubling, recursive halving algorithm, neighbor exchange
  - Linear, ring, gather + scatter
  - Rabenseifner's algorithm, Butterfly

Hardware-accelerated algorithms

- NVIDIA/Mellanox SHARP (Allreduce, Reduce, Barrier, Bcast)
- Mellanox multicast (Bcast)
- IBM BG tree network (Bcast)

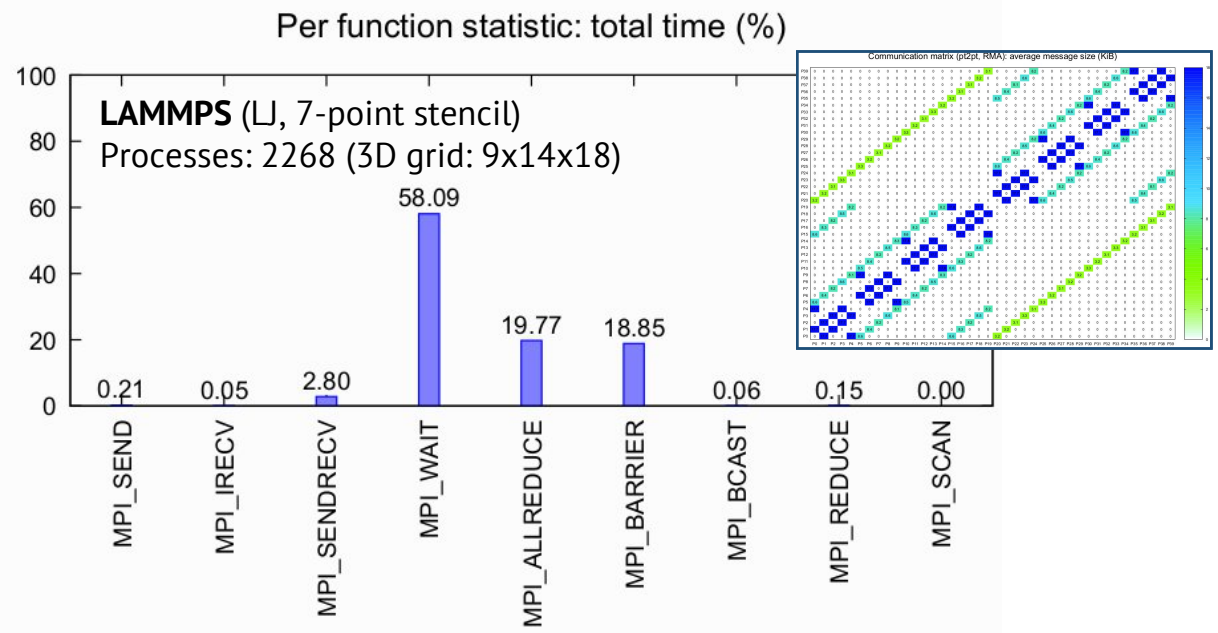
Topology-aware algorithms

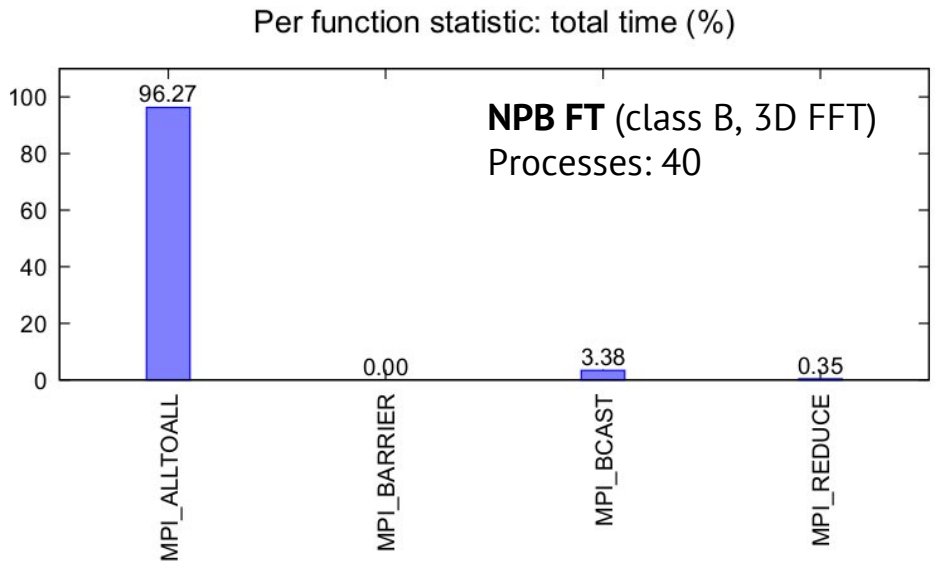
Topology of network is known
(routes, distances, process placement)

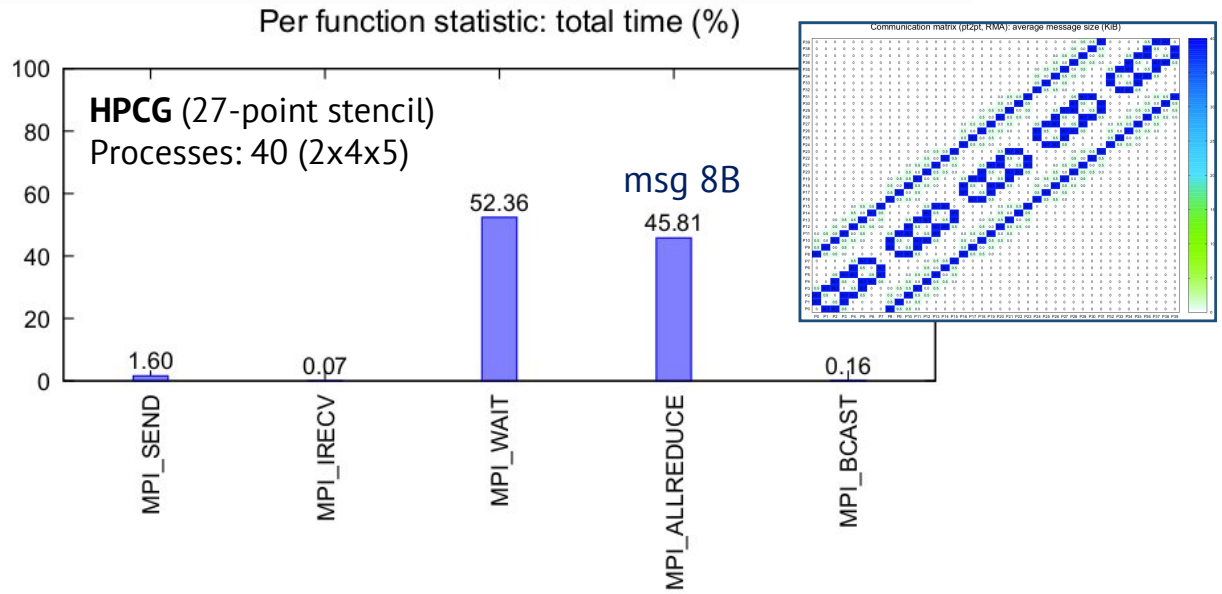
- Algorithms for Torus/Dragonfly networks
- Algorithms for Fat-tree topology
- Hierarchical collectives (SMP-aware): network + shared memory (intra-node)

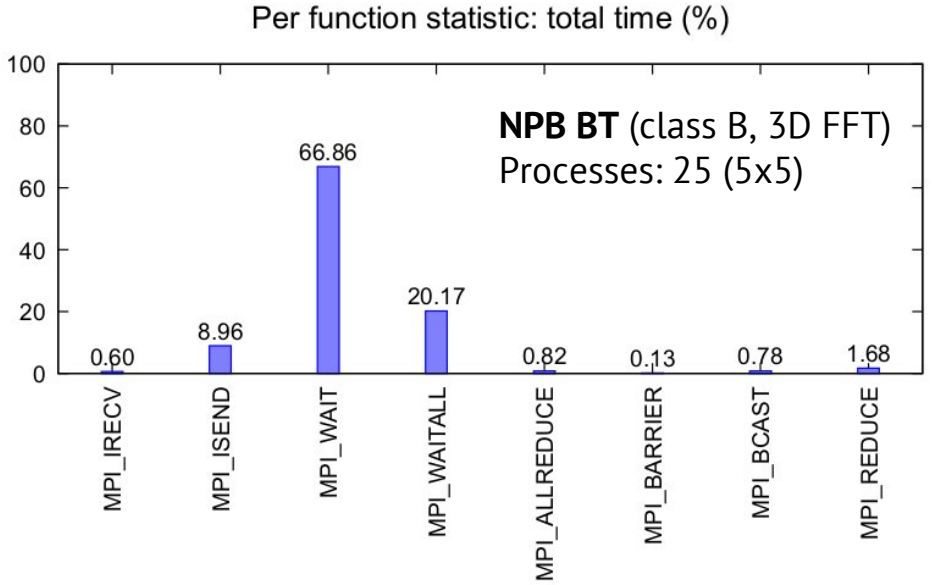
- MPI 4.0 Standard defines 17 (51) collective ops
- > 50 algorithms
(MPICH/MVAPICH, Open MPI, Intel MPI, HPC-X)

MPI Applications

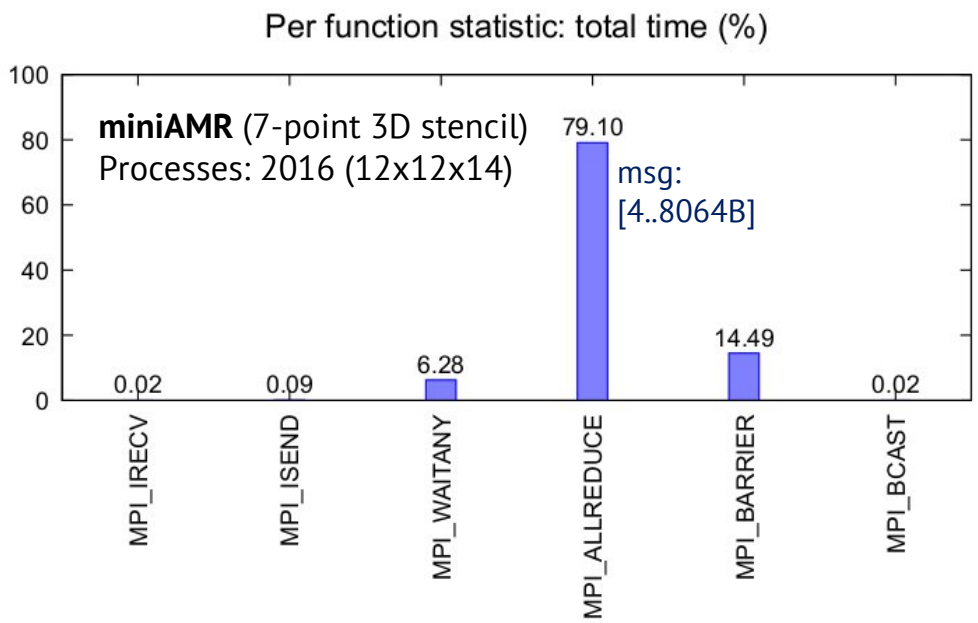


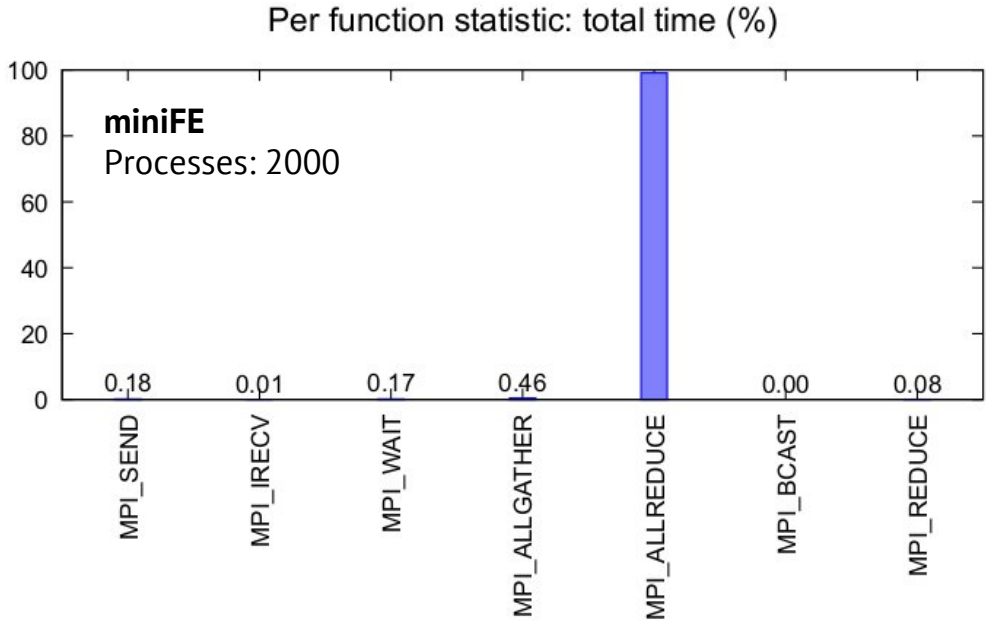


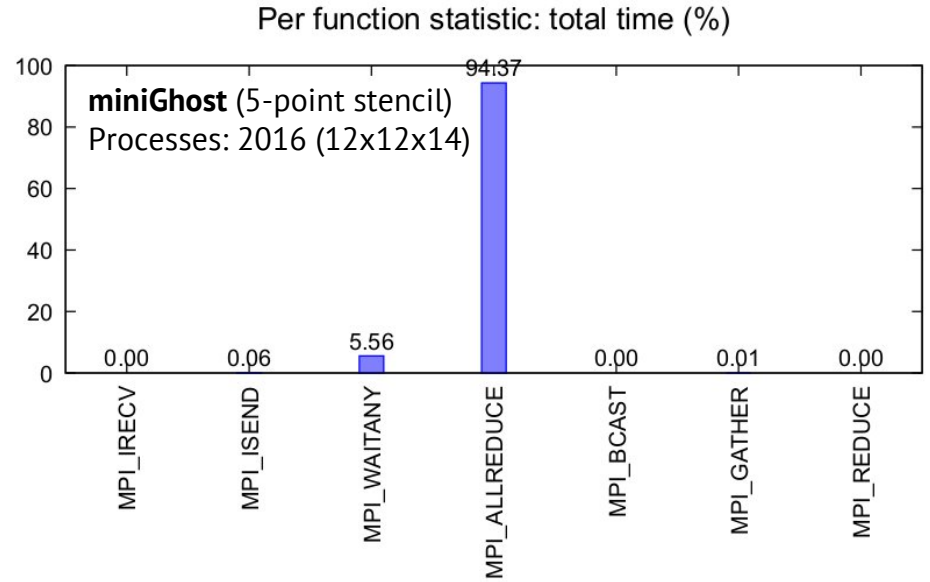


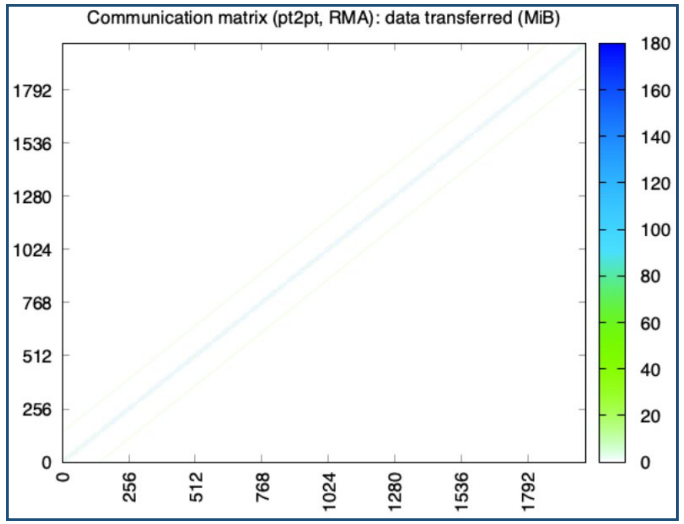


MPI Applications







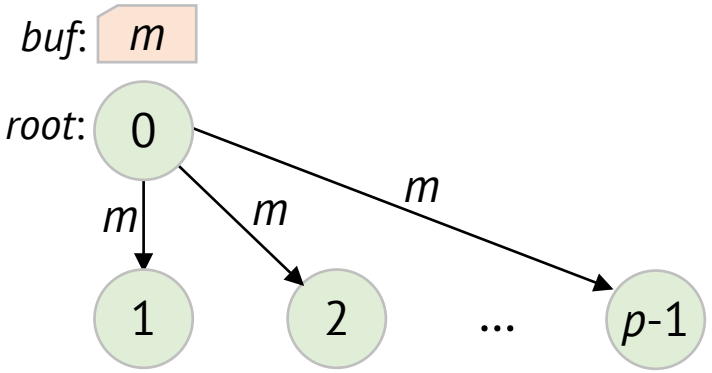


MPI_Bcast: broadcasts a message to all processes

```
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
```

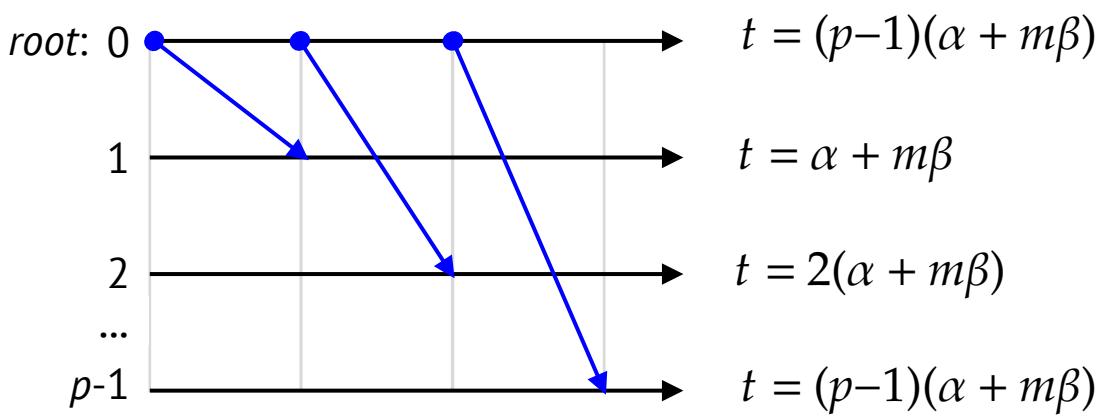
- **Linear broadcast (flat tree):** корневой процесс root передает сообщение остальным

Communication tree



$$t_{\text{Linear}} = (p-1)\left(\alpha + m\beta\right) = O(p)$$

Time diagram



- Плюсы: простота
- Минусы: плохая масштабируемость $O(p)$
- **Необходим более эффективный алгоритм**

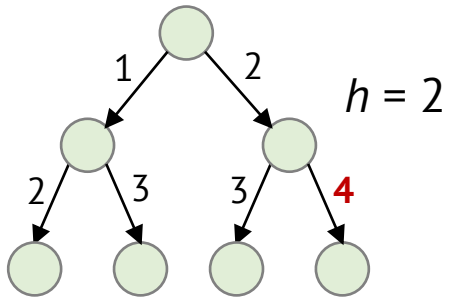
Cost model:

- p – number of processes
- α – latency (or startup time) per message
- β – transfer time per byte
- m – message size (bytes)

k-ary Tree Broadcast

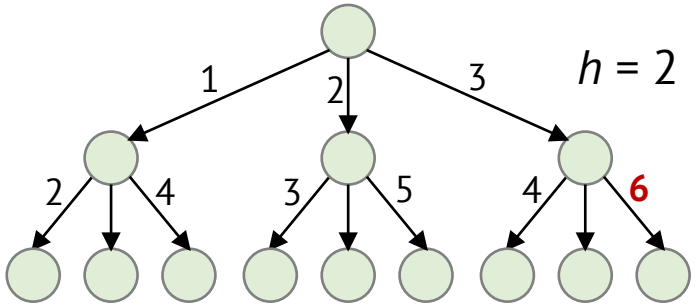
(*k* – node degree)

Binary tree (*k*=2, *p*=7)



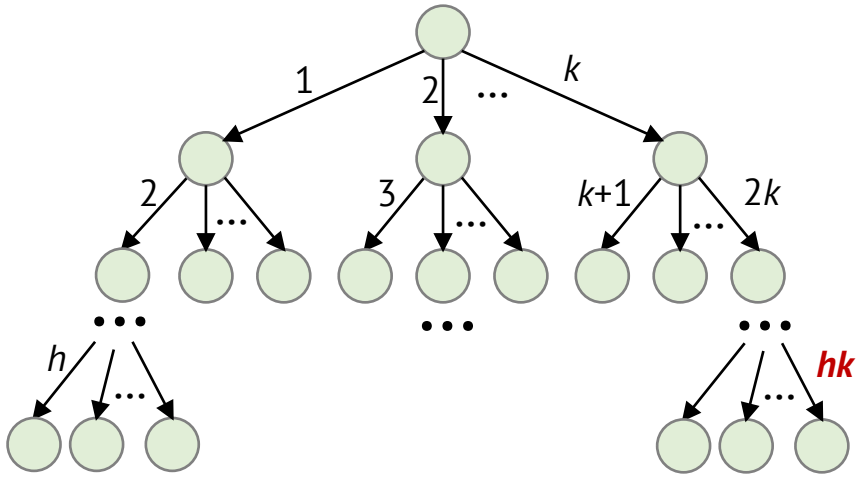
$$t = 4(\alpha + m\beta)$$

Ternary tree (*k*=3, *p*=13)



$$t = 6(\alpha + m\beta)$$

k-ary tree



$$t_{\text{Binary}} = 2\lceil \log_2 p \rceil (\alpha + m\beta)$$

$$t_{\text{Ternary}} = 3\lceil \log_3 p \rceil (\alpha + m\beta)$$

$$t_{\text{Kary}} = kh(\alpha + m\beta) = k\lceil \log_k p \rceil (\alpha + m\beta)$$

- Ключевая идея — параллельная передача сообщений в поддеревьях
- Какое дерево лучше?
- Как выбрать оптимальное значение *k* в модели latency-bandwidth?

k-ary Tree Broadcast

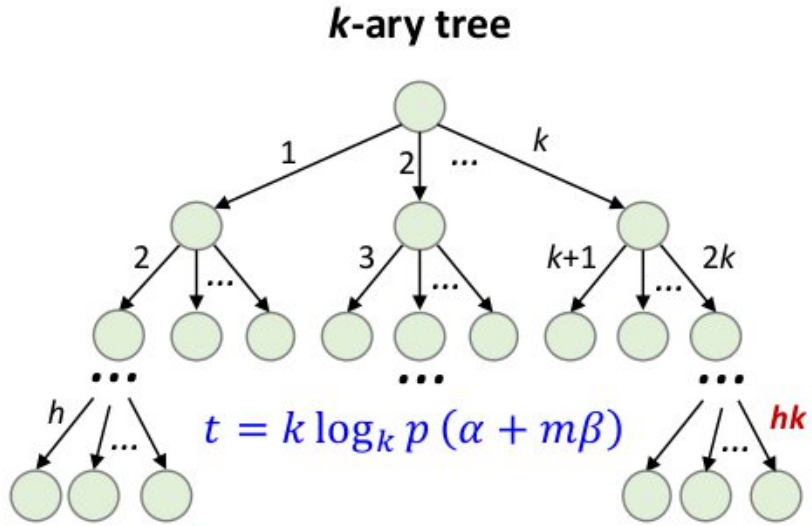
(*k* – степень узла в дереве)

- **Найдем оптимальное значение *k*** (степень узла)

$$t_{\text{Kary}}(k) = k \log_k p\,(\alpha + m\beta)$$

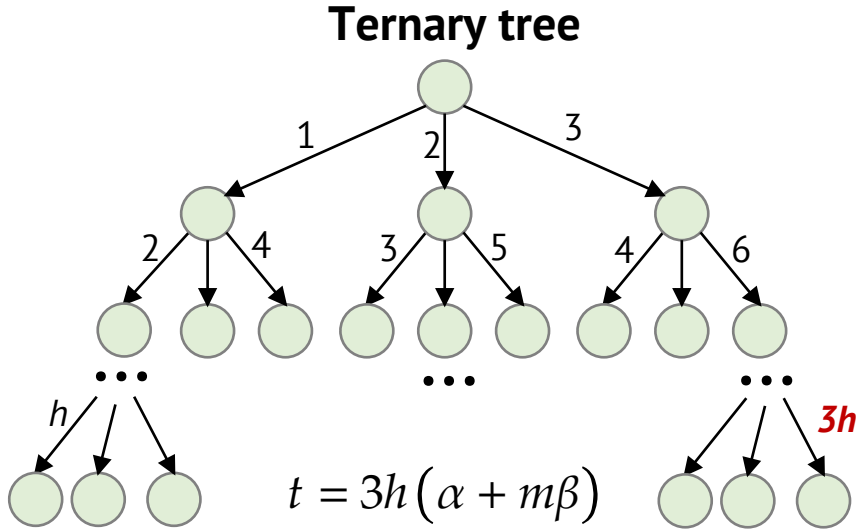
$$\frac{dt_{\text{Kary}}(k)}{dk} = \frac{\ln p \ln k - \ln p}{\ln^2 k} = 0$$

$$k_{\text{opt}} = e \approx 2.7183$$



- **Тернарное дерево (*k* = 3) является оптимальным в модели latency-bandwidth**

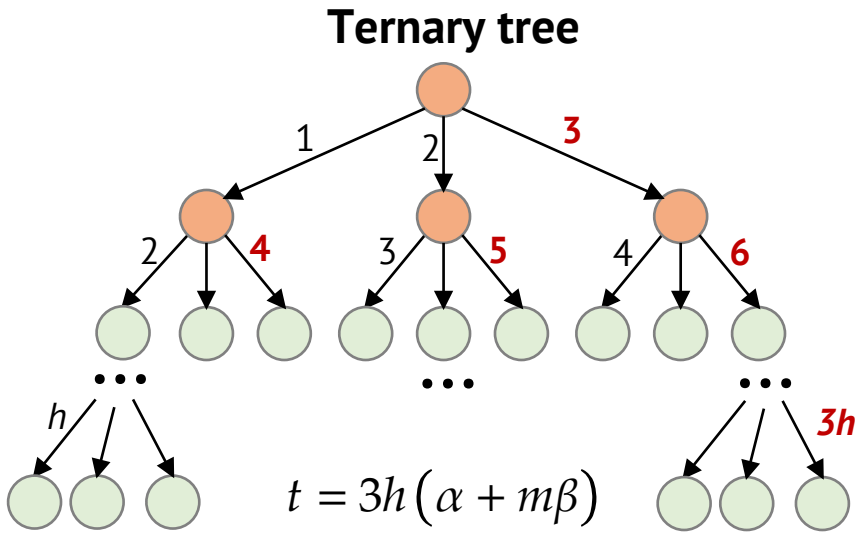
- Оптимально в рамках модели latency-bandwidth!
- Степень *k* не зависит от *p*, latency, bandwidth?



k-ary Tree Broadcast

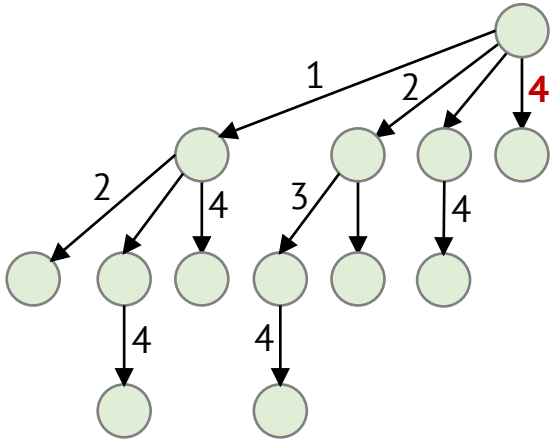
(*k* – степень узла в дереве)

- **Можно выполнить Bcast быстрее чем тернарное дерево?**
- Ограничение тернарного дерева — корень выбывает из операции после передачи *k* сообщений (const)
- Целесообразно выполнять передачу пока есть простаивающие потомки — использовать все доступные коммуникаицонные ресурсы



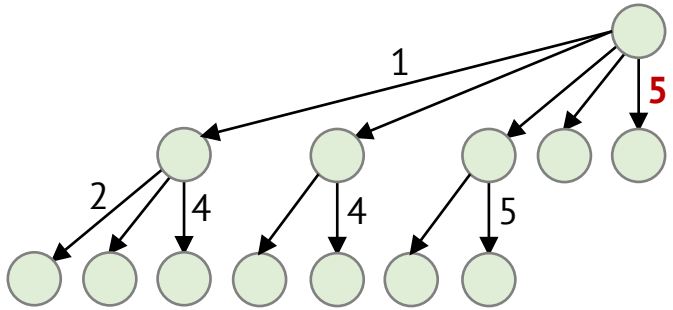
k-nomial Tree Broadcast

Binomial tree (*k*=2, *p*=13)



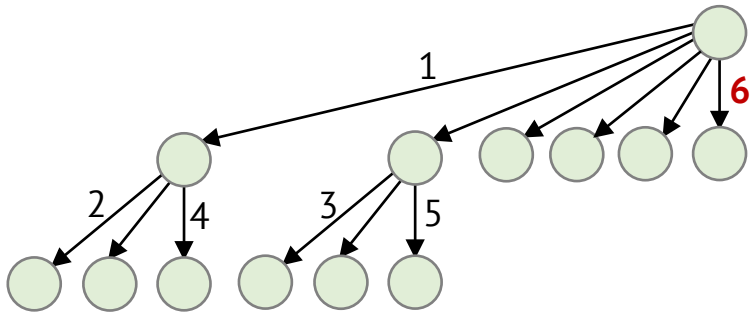
$$t = 4\left(\alpha + m\beta\right)$$

3-nomial tree (*p*=13)



$$t = 5\left(\alpha + m\beta\right)$$

4-nomial tree (*p*=13)



$$t = 6\left(\alpha + m\beta\right)$$

$$t_{\text{Binomial}} = \lceil \log_2 p \rceil \ \left(\alpha + m\beta\right)$$

$$t_{\text{3Nomial}} = \lceil \log_2 p \rceil \ \left(\alpha + m\beta\right)$$

$$t_{\text{KNomial}} = (k-1)\lceil \log_k p \rceil \ \left(\alpha + m\beta\right)$$

$$t_{\text{KNomial}} = (k-1)\lceil \log_k p \rceil \ \left(\alpha + m\beta\right) = O(\log p)$$

- Ключевая идея — степень дерева зависит от числа потомков в поддереве
- Найдем оптимальное значение *k* в модели latency-bandwidth

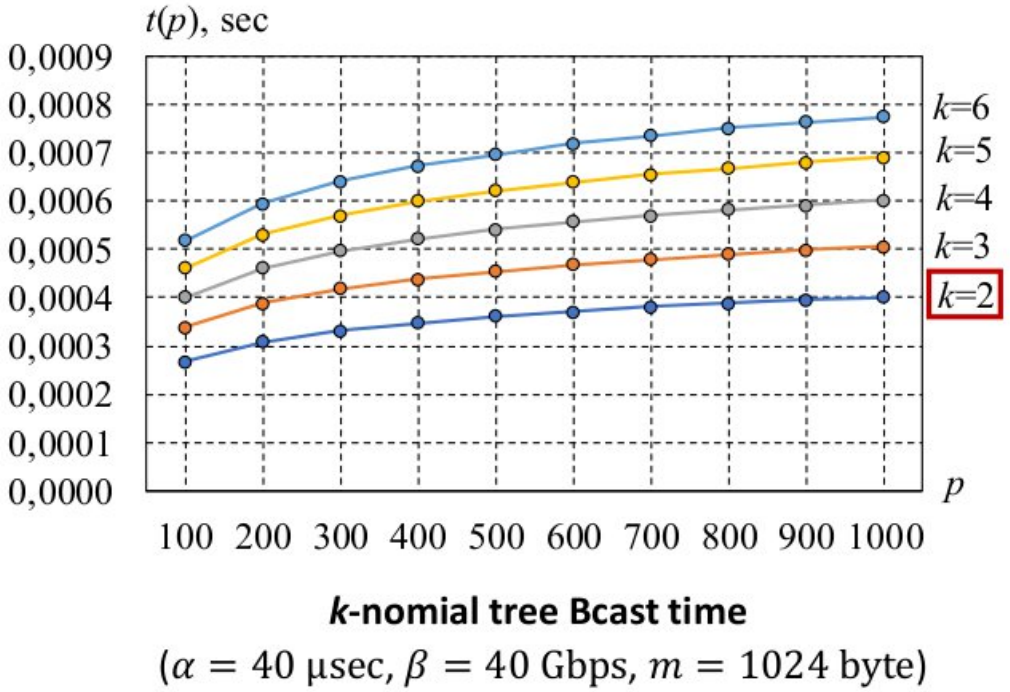
k-nomial Tree Broadcast

- **Найдем оптимальное значение *k* > 1**

$$t_{\text{KNomial}} = (k-1)\lceil \log_k p \rceil \ \ (\alpha + m\beta)$$

- **Время монотонно увеличивается с ростом *k***

$$k_{\text{opt}} = 2$$



***k*-nomial tree Bcast time**

($\alpha = 40$ µsec, $\beta = 40$ Gbps, $m = 1024$ byte)

- **Биномиальные деревья широко применяются на практике: Bcast, Reduce, Gather, Scatter**
  (Open MPI, MPICH, MVAPICH, Intel MPI)

- **Можно выполнить Bcast быстрее чем биномиальное дерево?**

Lower Bound on the Broadcasting Time

- **Нижняя граница (lower bound) времени выполнения Bcast [1, 2]:**

$$t_{\text{Bcast}} \geq \min\{\lceil \log_2 p \rceil \alpha, m\beta\}$$

$$t_{\text{Binomial}} = \lceil \log_2 p \rceil \alpha + \lceil \log_2 p \rceil m\beta$$

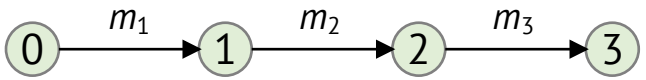
- **K-nomial tree асимптотически оптимально для сообщений небольших рамзеров (*m* = 1): $O(\log p)$**
- Binomial tree в $\lceil \log_2 p \rceil$ раз больше нижней границы по пропускной способности
- **Требуется более эффективный алгоритм для больших сообщений**

1. Sanders Peter, Speck Jochen, Träff Jesper Larsson. ***Two-tree algorithms for full bandwidth broadcast, reduction and scan*** // Parallel Computing. – 2009. – Vol. 35 (12). – pp. 581–594.
2. Hoefler T., Moor D. ***Energy, Memory, and Runtime Tradeoffs for Implementing Collective Communication Operations*** // Journal of Supercomputing Frontiers and Innovations. – 2014. – Vol 1 (2). – pp. 58-75
3. Thakur, R. Rabenseifner, W. Gropp. ***Optimization of collective communication operations in MPICH*** // Int. Journal of High Performance Computing Applications. – 2005. – Vol. 19 (1). – P. 49-66.

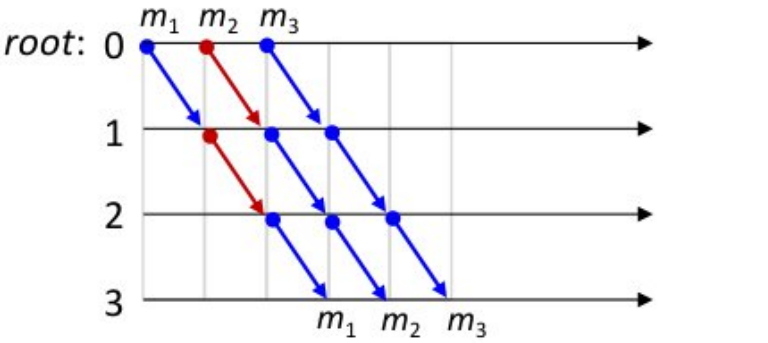
Linear Pipeline with Message Segmentation

(*p* small, *m* large)

- Сообщение *m* разбивается на сегменты по *s* байт: $m = (m_1, m_2, \ldots, m_{m/s})$
- Процесс *i* передает очередной сегмент процессу *i* + 1



$$t_{\text{Pipeline}} = \left(p - 2 + \frac{m}{s}\right)(\alpha + s\beta) = O(p)$$



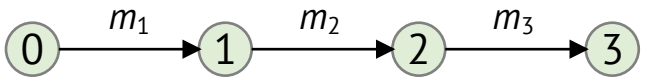
p = 4, 3 segments, **full duplex** channels

- **Определим оптимальный размер *s* сегмента в модели latency-bandwidth?**

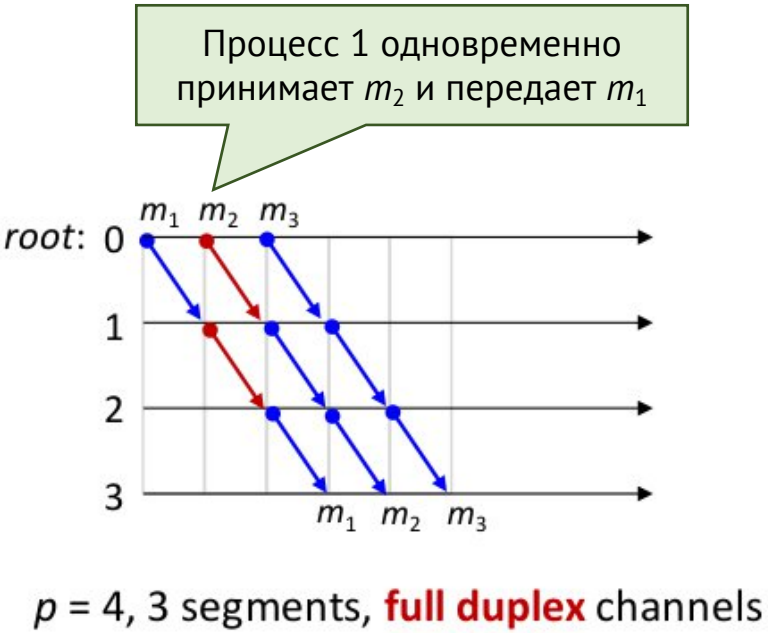
Linear Pipeline with Message Segmentation

(*p* small, *m* large)

- Сообщение *m* разбивается на сегменты по *s* байт: $m = (m_1, m_2, \ldots, m_{m/s})$
- Процесс *i* передает очередной сегмент процессу *i* + 1



$$t_{\text{Pipeline}} = \left(p - 2 + \frac{m}{s}\right)(\alpha + s\beta) = O(p)$$



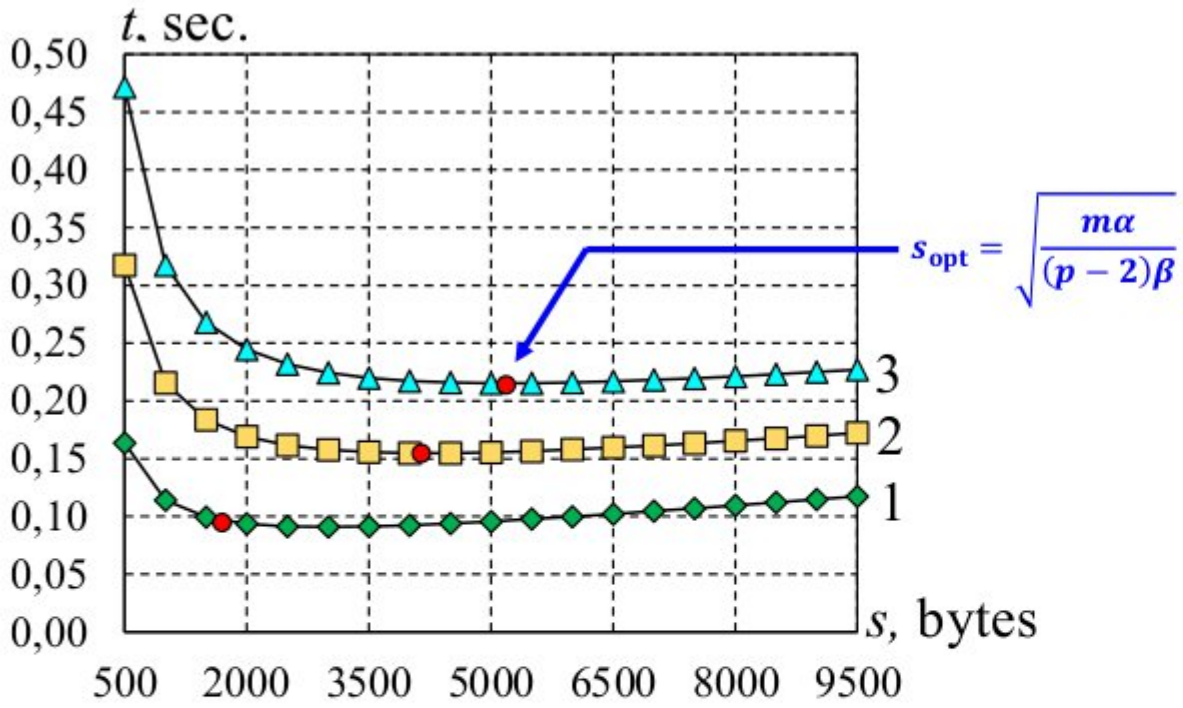
p = 4, 3 segments, **full duplex** channels

- **Определим оптимальный размер *s* сегмента в модели latency-bandwidth?**

$$\frac{dt(s)}{ds} = -\frac{m}{s^2}\alpha + (p-2)\beta = 0$$

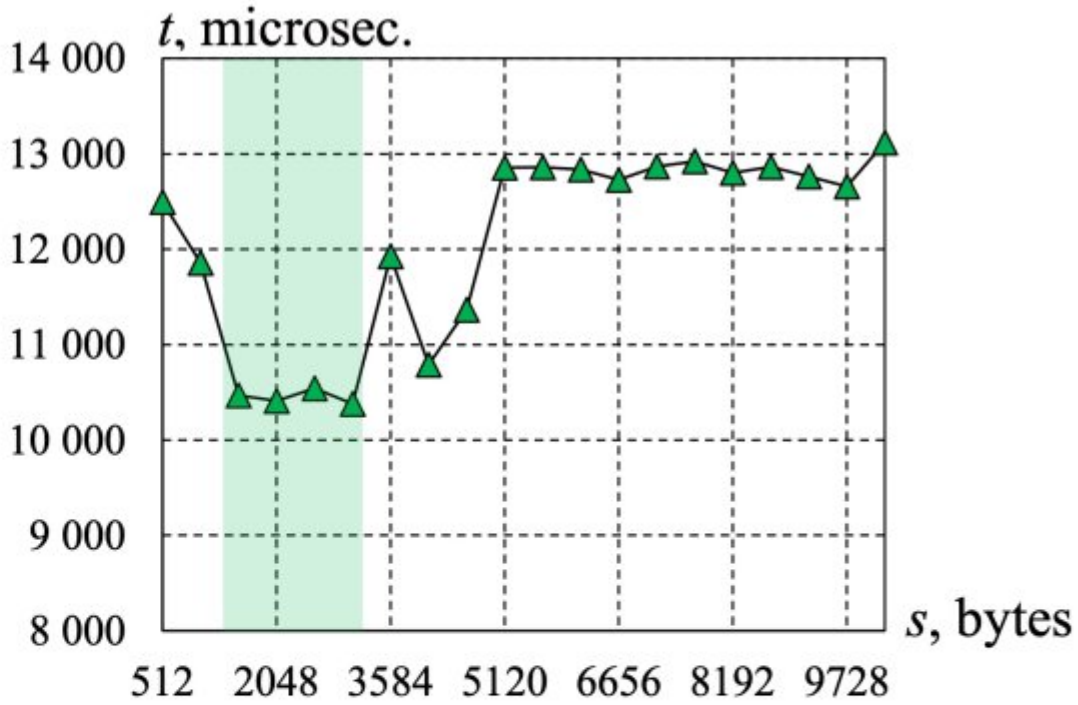
$$s_{\text{opt}} = \sqrt{\frac{m\alpha}{(p-2)\beta}}$$

Linear Pipeline with Message Segmentation



Latency-bandwidth model

(latency $\alpha = 5 \cdot 10^{-5}$ sec, bandwidth $\beta = 4.7 \cdot 10^{-8}$ sec)



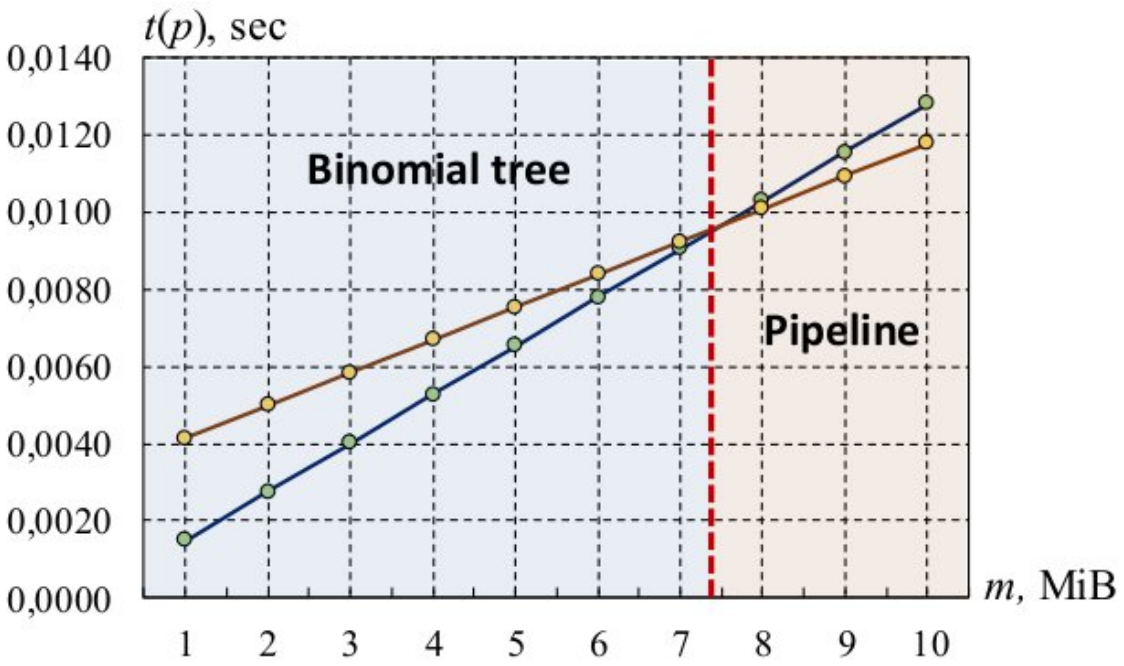
Gigabit Ethernet

(Open MPI 4.0.0, Intel MPI Benchmarks)

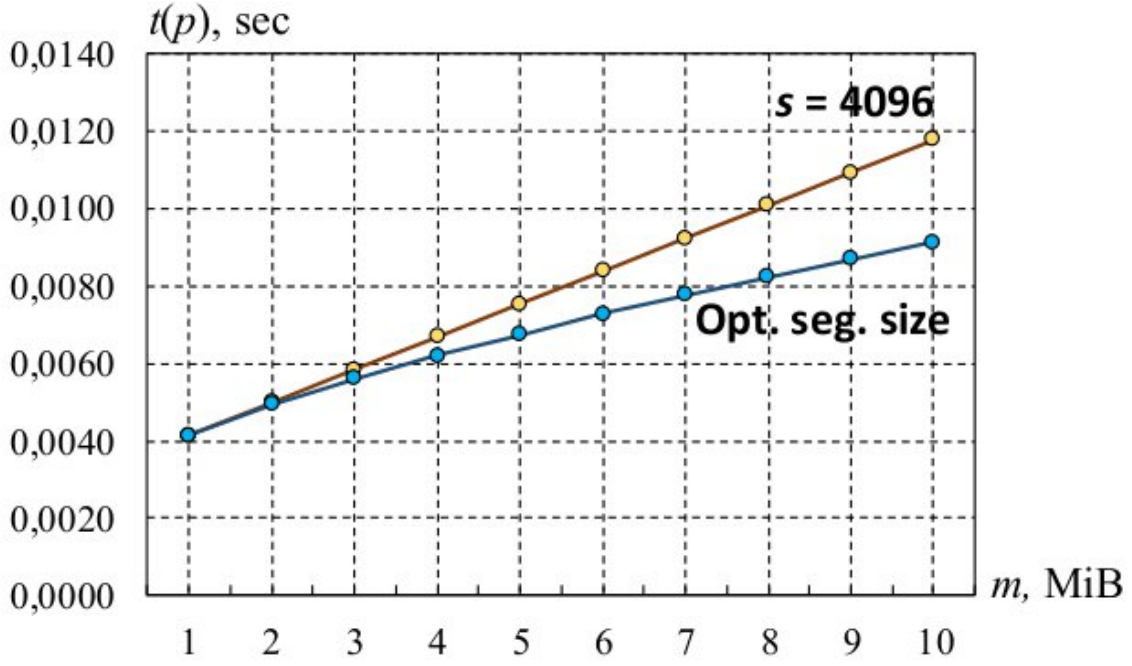
Linear pipeline algorithm time for various segment size *s* (128 processes):

1) *m* = 1 MiB; 2) *m* = 2 MiB; 3) *m* = 3 MiB

Binomial tree vs. Pipeline



Segment size: $s = 4096$ **bytes,**
$\alpha = 40$ μsec, $\beta = 40$ Gbps, $p = 64$



Optimal segment size,
$\alpha = 40$ μsec, $\beta = 40$ Gbps, $p = 64$

- **Open MPI, MPICH, MVAPICH**
  - ❑ Binomial tree – small sized messages
  - ❑ Pipeline – large messages and small number of processes

Pipeline is a factor of $p/\log_2 p$ slower in latency (related to t_{Bcast})

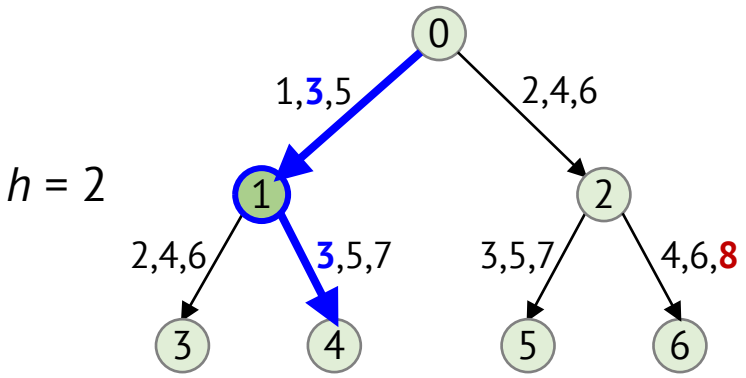
- **What can we do for intermediate message sizes?**

Pipelined Binary Tree

(combine pipeline and tree)

Segmented message: $m = (m_1, m_2, m_3)$

$p = 7$, $root = 0$



	Time step							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Send-Recv operations	0-1	0-2 1-3	0-1 1-4 2-5	0-2 2-6 1-3	0-1 1-4 2-5	0-2 2-6 1-3	1-4 2-5	2-6

- At step 3 process 1 recvs from 0 and sends to 4 simultaneously (full duplex model)

$$t_{\text{PipelinedTree}} = 2\left(h + n - 1\right)\left(\alpha + s\beta\right) = 2\left(\lceil \log_2 p \rceil + \frac{m}{s} - 1\right)\left(\alpha + s\beta\right)$$

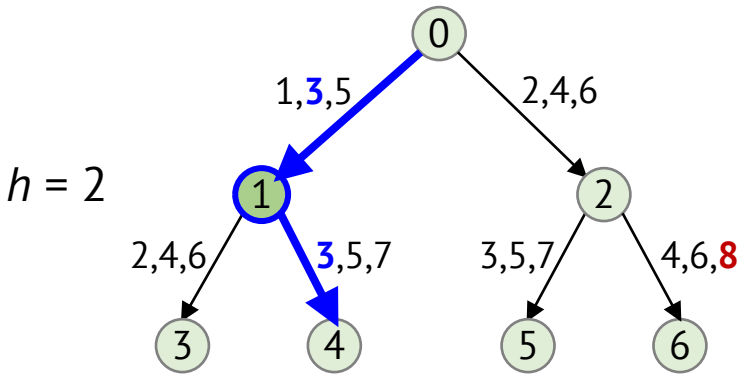
- $n = m/s$ – number of segments
- **What is the optimal segment size *s* in latency-bandwidth model?**

Pipelined Binary Tree

(combine pipeline and tree)

Segmented message: $m = (m_1, m_2, m_3)$

$p = 7$, $root = 0$



	Time step							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Send-Recv operations	0-1	0-2 1-3	0-1 1-4 2-5	0-2 2-6 1-3	0-1 1-4 2-5	0-2 2-6 1-3	1-4 2-5	2-6

- At step 3 process 1 recvs from 0 and sends to 4 simultaneously (full duplex model)

$$t_{\text{PipelinedTree}} = 2(h + n - 1)(\alpha + s\beta) = 2(\lceil \log_2 p \rceil + \frac{m}{s} - 1)(\alpha + s\beta)$$

- $n = m/s$ – number of segments
- **What is the optimal segment size *s* in latency-bandwidth model?**

$$\frac{dt(s)}{ds} = -\frac{2m}{s^2}\alpha + 2(\log_2 p - 1)\beta = 0$$

$$s_{\text{opt}} = \sqrt{\frac{m\alpha}{(\log_2 p - 1)\beta}}$$

Pipelined Binary Tree

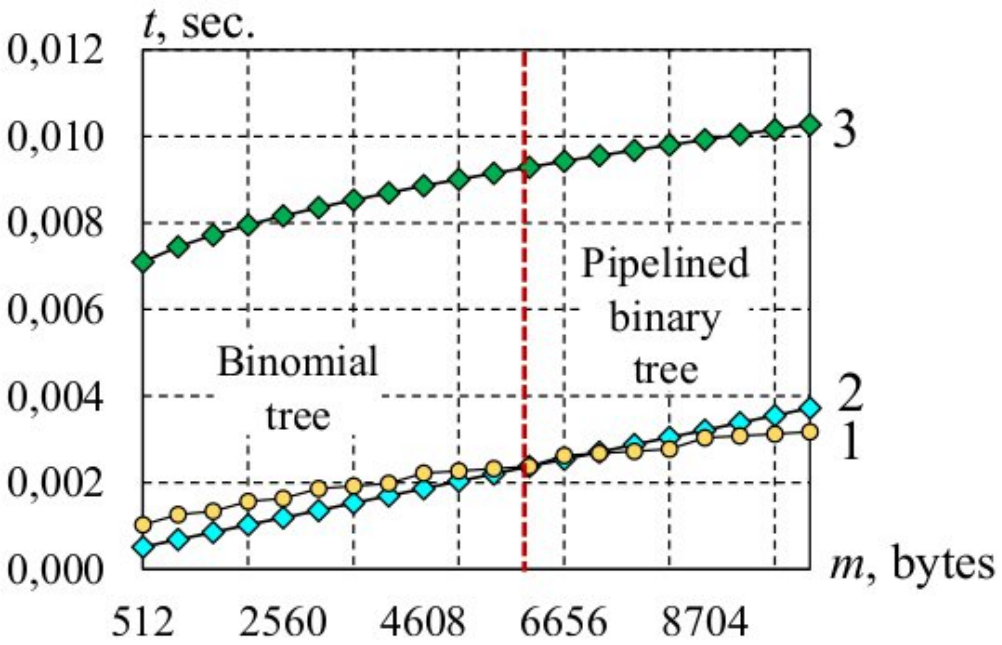
(combine pipeline and tree)

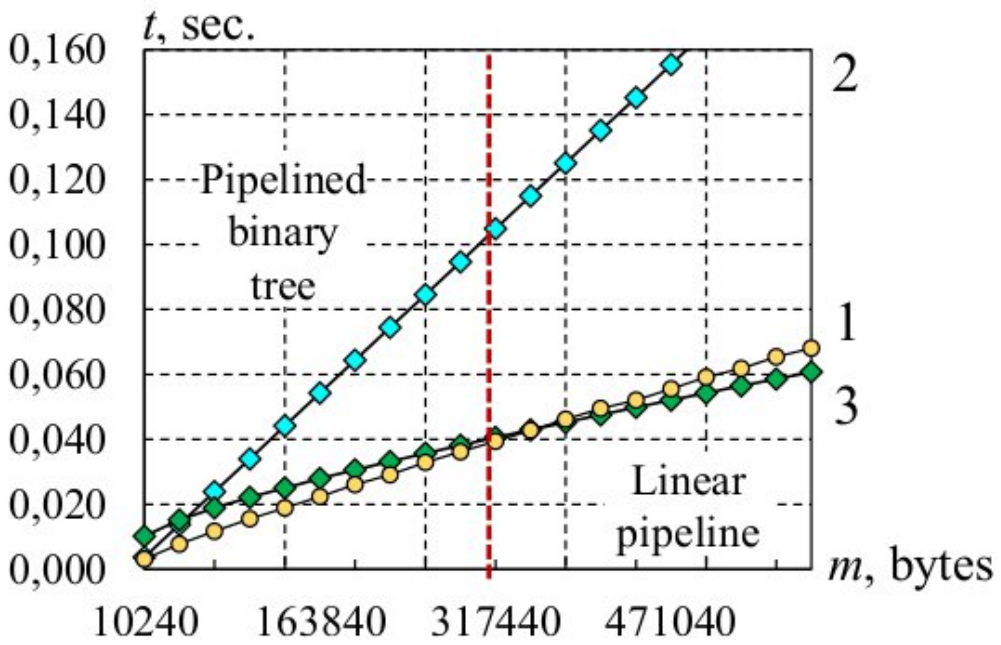
$$t_{\text{Bcast}} \geq \min\{\lceil \log_2 p \rceil \alpha, m\beta\}$$

$$t_{\text{PipelinedTree}} = \mathbf{2}\left(\lceil \log_2 p \rceil + \frac{m}{s} - 1\right)(\alpha + s\beta)$$

- Small messages ($m = 1$, $s = 1$), large p: $O(\log p)$
- Large messages, p, α, β – constants: $O(m\beta)$
- For large p and m bandwidth is off by a factor of 2

MPI_Bcast





Execution time of MPI_Bcast (model) :

(α = 0.00005 s, β = 0.000000047 s, p = 128, *root* = 0):

1 – pipelined binary tree (opt. segment); 2 – binomial tree; 3 – linear pipeline (opt. segment)

Bandwidth-Optimal Broadcast

[*] Sanders Peter, Speck Jochen, Träff Jesper Larsson. ***Two-tree algorithms for full bandwidth broadcast, reduction and scan*** // Parallel Computing. – 2009. – Vol. 35 (12). – pp. 581–594.

- **Key ideas**:

❑ In binomial tree, all leaves only receive data and never send

❑ Send along two simultaneous binary trees where the leaves of one tree are inner nodes of the other

Allreduce

Алгоритм	Время выполнения	Ограничения
Recursive doubling	$2\log_2(p)\alpha + 2\log_2(p)m\beta + \log_2(p)m\gamma$	
ReduceScatter + Allgather (Rabenseifner)	$\alpha\log_2(p) + m\beta + m\gamma$	коммутативные операции
Ring	$(p-1)(2\alpha + 2m\beta + m\gamma)$	коммутативные операции
ReduceScatter ring + Allgather ring	$(p-1)(4\alpha + 4m/p\beta + m\gamma)$	коммутативные операции

Allreduce: ReduceScatter + Allgather

- **Алгоритм Р. Рабенсейфнера**
- Ограничения:
  - $count \geqslant 2^{\lfloor \log 2(p) \rfloor}$
  - p — степень числа 2
  - только коммутативные операции

Этап 1. Переход к числу процессов, равному степени двойки — активными остаются $2^{\lfloor \log 2(p) \rfloor}$ процессов

Этап 2. ReduceScatter — На каждом из $\log_2(p')$ шагов размер передаваемого сообщения уменьшается в два раза (vector halving), а расстояние между взаимодействующими процессами увеличивается в два раза (distance doubling)

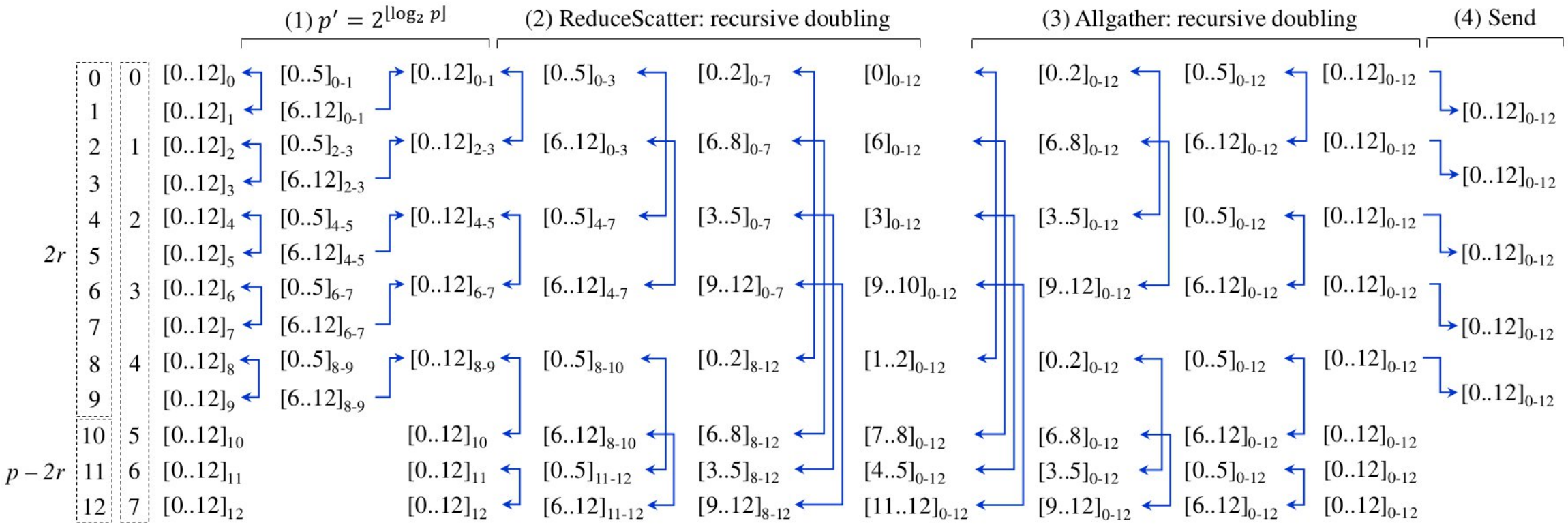
Этап 3. Allgather. Результаты частичной редукции передаются всем процессам операцией Allgather (recursive doubling)

Этап 4. Передача результата неактивным процессам

Allreduce: ReduceScatter + Allgather

Алгоритм Р. Рабенсейфнера

$count \geqslant \text{count} \geqslant 2^{\lfloor \log_2(p) \rfloor}, \quad t_{\text{Allreduce}} = O\left(m\lambda + \alpha \log p + m\beta + m\gamma\right)$



Этап 1. Переход к числу процессов, равному степени двойки — активны $2^{\lfloor \log_2(p) \rfloor}$ процессов

Этап 2. ReduceScatter — на каждом из $\log_2 p'$ шагов размер сообщения уменьшается в два раза (vector halving), а расстояние между процессами увеличивается в два раза (distance doubling)

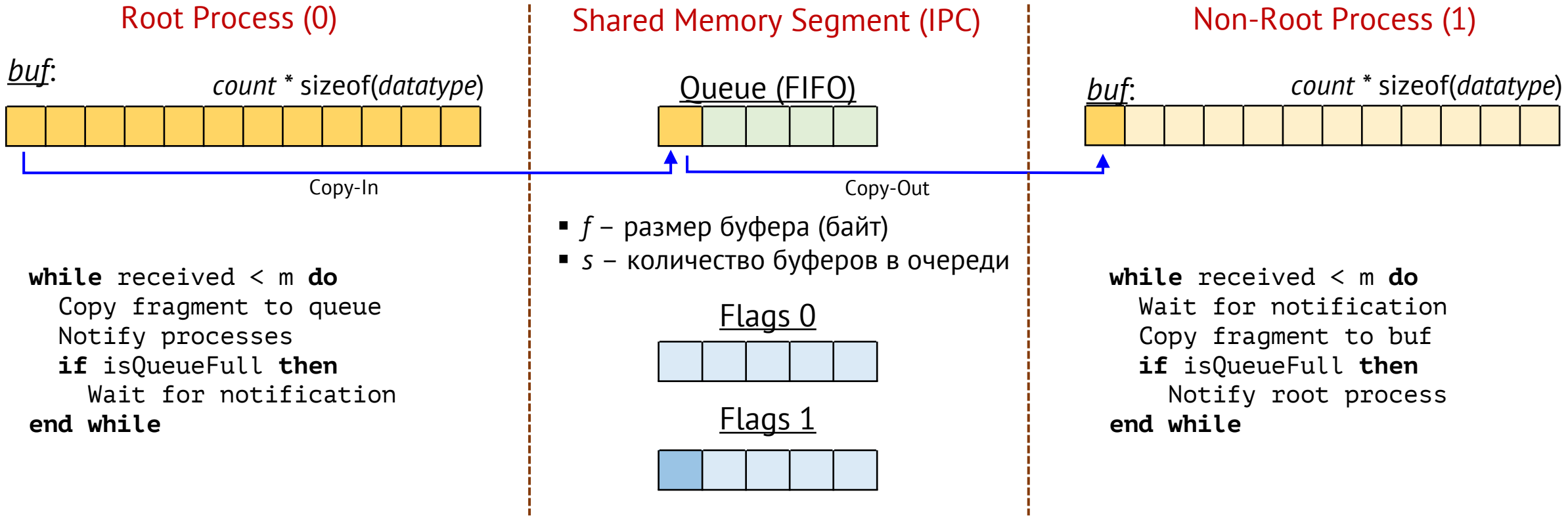
Этап 3. Allgather. Результаты частичной редукции передаются всем процессам операцией Allgather (recursive doubling)

Shared Memory Bcast

I) Семейство CICO-алгоритмов (Copy-In/Copy-Out): Open MPI, MVAPICH, Intel MPI

1. При формировании коммуникатора создается сегмент разделяемой памяти и система очередей в нем
2. При вызове MPI_Bcast корневой процесс выполняет конвейерную передачу сообщения через очередь

```
MPI_Bcast(buf, count, datatype, root, comm)
```



II) Семейство ZeroCopy-алгоритмов (одно копирование): KNEM (Linux), XPMEM (Cray, Linux), CMA (Linux)

MVAPICH

- **Циклическая очередь из *w* (128) слотов для каждого процесса**
  - ❑ Буфер длины *f* байт (8192)
  - ❑ Номер *psn* операции, на которой слот заполнен
- Некорневые процессы ждут пока корень не установит поле *psn* в значение *read*
- Если все слоты заполнены, осуществляется барьерная синхронизация
- Размер сегмента: *O*(*pwf*)
- Хранение указателей на слоты: *O*(*pw*)
- Уведомление процессов: flat tree *O*(*p*)
- Топология NUMA-системы при размещении очередей в памяти не учитывается

Shared Memory Region

	0	1	…	window_size – 1
0	psn: uint32 tail_psn: uint32 buf[8192]: uint8			
1				
…				
p – 1				

Per process: queue[p][wsize]

	tail=0	write=1 read=1		
	0	1	…	window_size – 1
0	*ptr to shm*			
1				
…				
p – 1				

http://mvapich.cse.ohio-state.edu/ (OFA-IB-CH3)

Обзор работ: Open MPI coll/sm

- **Для каждого процесса:**
  - ❑ Очередь из *w* (8) буферов по *f* байт (8KB)
  - ❑ *w* блоков с управляющей информацией

- Барьерная синхронизация реализована на глобальных счетчика и атомарной операции
- Очередь разбита на два множества (banks)

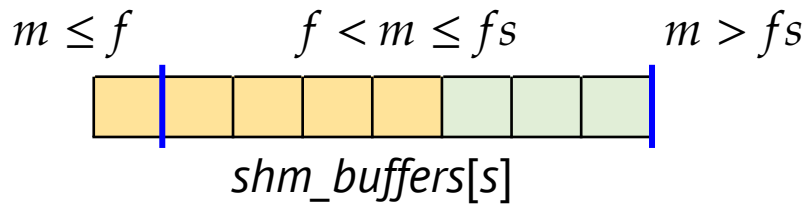
- Уведомление процессов и копирование фрагментов:
  k-арное дерево (complete *k*-ary tree)

- Размер сегмента: $O(pwf + pc)$
- Хранение указателей на буферы и дерево: $O(w + pk)$.

Proc	T	Queues (buffers, controls)							
		Set 1				*Set* 2			
		▪ **nprocs_using**: uint32 ▪ **op**: uint32				▪ **nprocs_using**: uint32 ▪ **op**: uint32			
0	Q:	F0	F1	F2					
	C:								
1	Q:								
	C:	S0	S1	S2					
2	Q:								
	C:	S0	S1	S2					
...									
p – 1	Q:								
	C:	S0	S1	S2					

- [SM08] *Graham R.L., Shipman G.* **MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives** // Proc. of the 15th European PVM/MPI Users' Group Meeting, 2008
- [INTEL18] *Jain S., Kaleem R., Balmana M., Langer A., Durnov D., Sannikov A. and Garzaran M.* **Framework for Scalable Intra-Node Collective Operations using Shared Memory** // Proc. of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC-2018)

Режимы работы очереди из *s* буферов



- При $m \le f$ конвейеризация передачи сообщения не используется (низкая эффективность)

$$t(m, s) = t_{WO} + t_A + t_W + 2mt$$

- При $f < m \le b$ включается конвейеризация передачи фрагментов сообщения, *s* буферов очереди вмещают всё сообщение (потери на ожидания минимальны, высокая эффективность)

$$t(m, s) = t_{WO} + t_A + ft + \lceil m/f \rceil t_W + mt$$

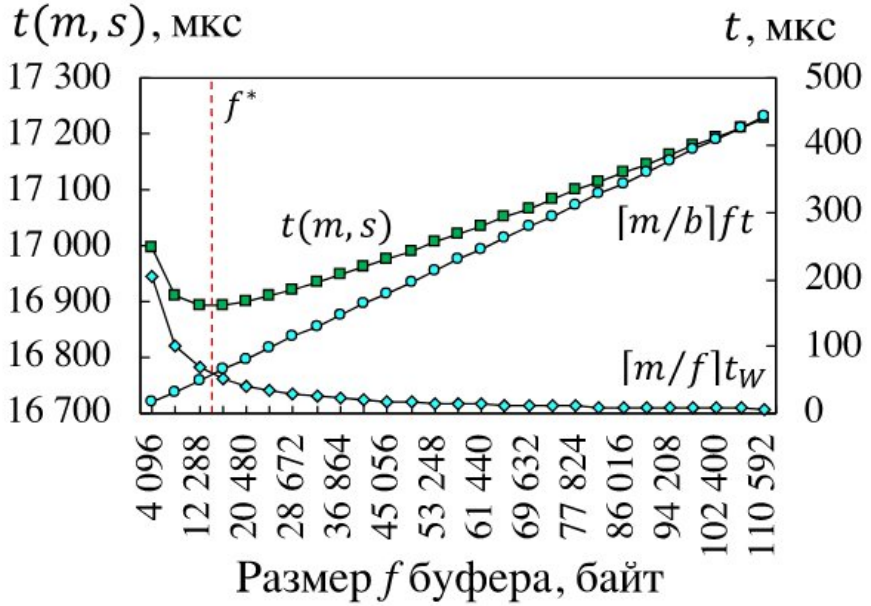
- При $m > b$ буферы очереди используются многократно, суммарное время ожидания возрастает с увеличением *m* и убывает с ростом *b*

$$t(m, s) = \lceil m/b \rceil (t_{WO} + t_A + ft) + \lceil m/f \rceil t_W + mt$$

Оптимальные параметры очереди

- **Найдем оптимальные** размеры f буферов и длины s очереди, которая помещается в b байт памяти и обеспечивает минимум времени выполнения алгоритма
- Например, определить оптимальную конфигурацию очереди, которая помещается в 1% от размера памяти, приходящийся на одно процессорное ядро

$$t(m,s) = \lceil m/b \rceil (t_{WO} + t_A) + \lceil m/b \rceil ft + m/f \cdot t_W + mt, \qquad \frac{\partial t}{\partial f} = -mt_W/f^2 + \lceil m/b \rceil t = 0,$$



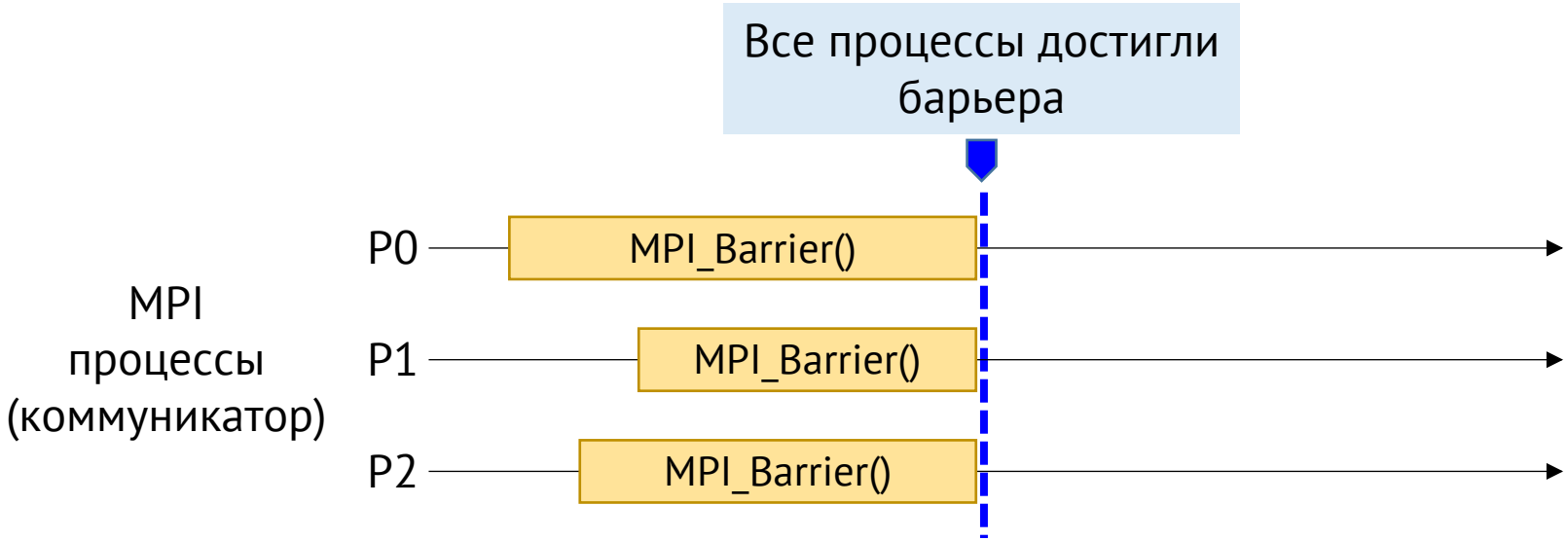
$$f^* = \sqrt{m/\lceil m/b \rceil \cdot t_W/t} \approx \sqrt{b \cdot t_W/t}$$
$$s^* = b/f^* = \sqrt{b \cdot t/t_W}$$

- Суммарное время $\lceil m/b \rceil ft$ ожидания – линейно возрастает с ростом f
- Суммарное время ожидания уведомлений $\lceil m/f \rceil t_W$ – убывает обратно пропорционально f

m = 16 MB, b = 4 MB, $s = b/f$, q = 1, $t = 10^{-9}$ с, $t_{WO} = 100t$, $t_A = 10t$, $t_W = 50t$

Барьерная синхронизация

- **Барьер** – операция синхронизации потоков/процессов, блокирующая их выполнение до тех пор, пока все потоки/процессы не достигнут барьера
- **Примеры**: MPI MPI_Barrier(), OpenMP #pragma omp barrier,
  POSIX pthread_barrier_wait(), C++20 std::barrier<>
- **Применение**: ожидание готовности данных на разделяемом ресурсе (в памяти, на внешнем носителе)



Алгоритмы барьерной синхронизации Open MPI

Алгоритмы на основе операций Send/Recv
Open MPI coll/tuned (base)

Алгоритм	Сложность
1. Linear	$O(p)$
2. Double ring	$O(p)$
3. Bruck	$O(\log(p))$
4. Recursive doubling	$O(\log(p))$
5. Tree (top/down recursive doubling)	$O(\log(p))$

Алгоритмы на базе сегмента разделяемой памяти [*]

Алгоритм	Сложность
1. Central Counter (CC)	$O(p)$ atomic ops
2. Flat tree (FT)	$O(p)$ memory ops
3. Flat tree Gather/Release (GR)	$O(p)$ memory ops
4. Combining Tree (CT)	$O(k\log_k(p))$ memory ops
5. MCS Barrier	$O(k\log_k(p) + q\log_q(p))$
6. Tournament (TR)	$O(\log(p))$
7. Dissemination (DS)	$O(\log(p))$

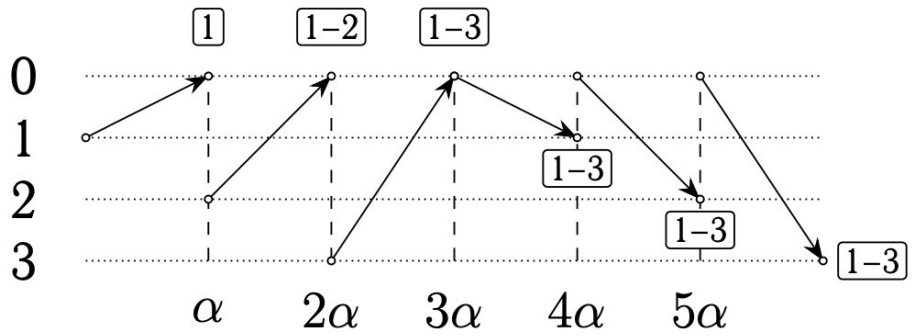
[*] Kurnosov M., Tokmasheva E. ***Optimizing Barrier Algorithms on Asymmetric Subsystems of NUMA Machines*** // Proc. of the IEEE Ural Symposium on Biomedical Engineering, Radioelectronics and Information Technology (USBEREIT-2021), 2021

Алгоритм с центральным счетчиком (linear, central counter)

Алгоритм 3.1. BARRIERCENTRALCOUNTER

```
1   if rank = 0 then
2       for i = 1 to p − 1 do          ▷ Gather: прием уведомлений от всех процессов
3           RECV(i)
4       end for
5       for i = 1 to p − 1 do          ▷ Bcast: передача разрешения на выход из барьера
6           SEND(i)
7       end for
8   else
9       SEND(0)                        ▷ Передача «пустого» сообщения процессу 0
10      RECV(0)
11  end if
```

- Время выполнения алгоритма
  $T = 2\alpha(p - 1) = O(p)$

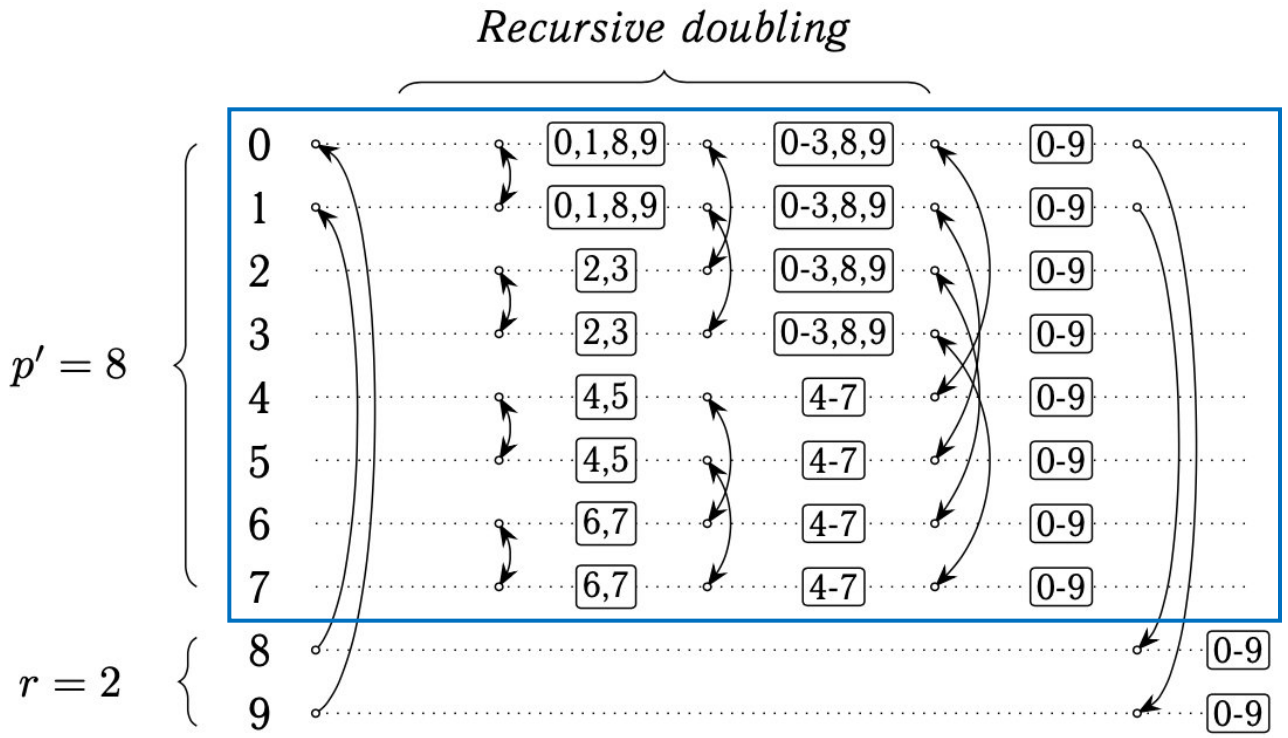


https://github.com/open-mpi/ompi/blob/01d5a86c6021a9610b512e06679c11ba8974a833/ompi/mca/coll/base/coll_base_barrier.c#L346

Алгоритм рекурсивного удваивания (recursive doubling)

```
11  dist = 1
12  while dist < p' do                ▷ Обмены рекурсивным удваиванием
13      peer = rank xor dist
14      if peer < p' then
15          SendRecv(peer, peer)
16      end if
17      dist = 2 · dist
18  end while
```

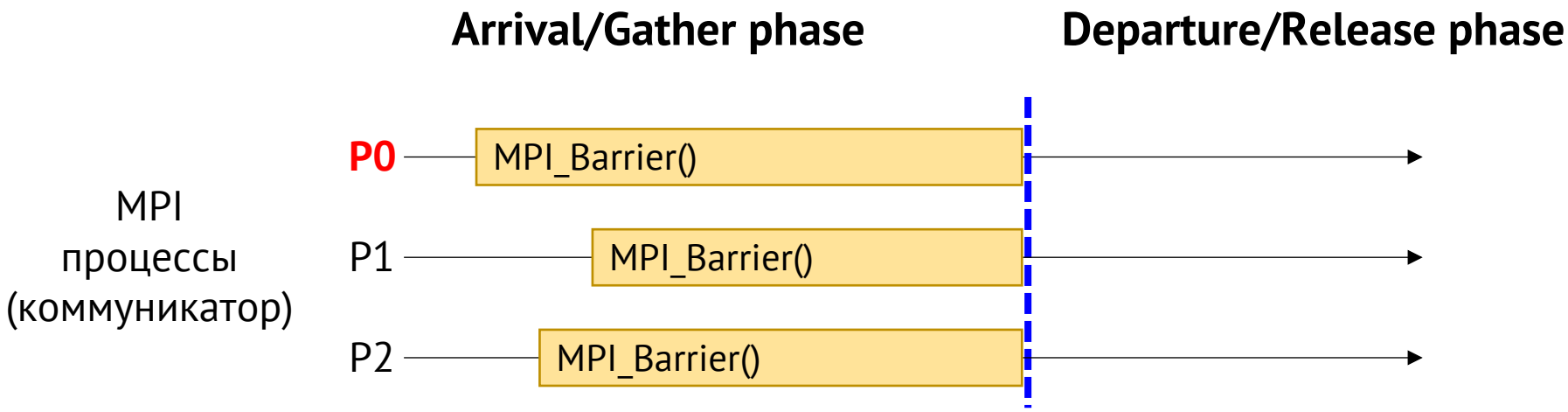
- Время выполнения алгоритма
  $T = 2\alpha + 2\alpha\lfloor \log_2 p \rfloor = O(\log(p))$



Алгоритмы на базе разделяемой памяти

Централизованные алгоритмы	Сложность
1. Central Counter (CC)	$O(p)$ atomic ops
2. Flat tree (FT)	$O(p)$ memory ops
3. Flat tree Gather/Release (GR)	$O(p)$ memory ops
4. Combining Tree (CT)	$O(k\log_k(p))$ memory ops
5. MCS Barrier	$O(k\log_k(p) + q\log_q(p))$
6. Tournament (TR)	$O(\log(p))$

Децентрализованные алгоритмы	Сложность
Dissemination (DS)	$O(\log(p))$



Central Counter Barrier (CC)

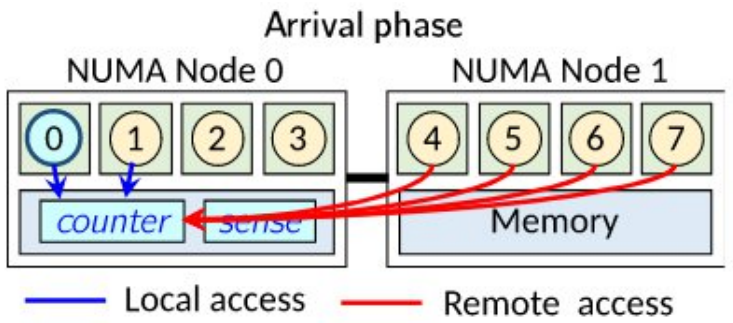
```
algorithm BarrierCentralCounter()
    if p < 2 then
        return;
    end if
    /*
     * Initial state:
     *   shm_counter = p
     *   sense = 1
     *   sense_local = 1 (per process)
     */
    sense_local = sense_local ^ 1

    if atomic_fetch_dec(shm_counter) = 1 then
        // Last process releases all
        shm_counter = p
        shm_sense = sense_local
    else
        while shm_sense != sense_local do
          // Wait all
        end while
    end if
end algorithm
```

Central Counter Barrier (CC)

- **Arrival**: каждый процесс атомарно уменьшает счетчик *counter*
- **Departure**: последний процесс выставляет *sense* = *sense_local*[*rank*]

Время: $O(p)$ атомарных операций

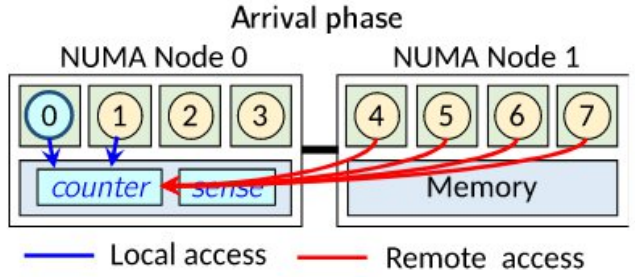


Алгоритмы на базе разделяемой памяти

Central Counter Barrier (CC)

- **Arrival**: каждый процесс атомарно уменьшает счетчик *counter*
- **Departure**: последний процесс выставляет *sense* = *sense_local*[*rank*]

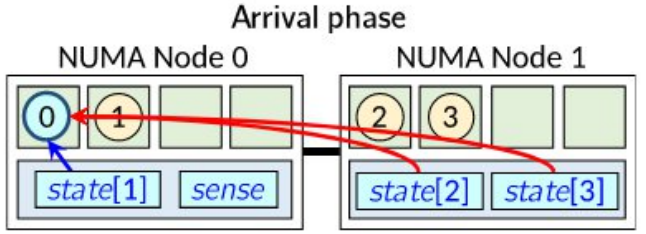
Время: *O*(*p*) атомарных операций



Flat Tree Barrier (FT)

- **Arrival**: корень ожидает дочерние процессы *child* = 0, 1, ..., *p* - 1: *state*[*child*] = *state*[*root*]
- **Departure**: корень выставляет *sense* = *sense_local*[*root*]

Время: *O*(*p*) операций чтения/запись



Flat Tree Gather/Release (GR)

- **Arrival**: корень ожидает дочерние: *gather_state*[*child*] = *gather_state*[*root*]
- **Departure**: корень уведомляет дочерние: *release_state*[*child*] = *release_state_local*[*root*]

Время: *O*(*p*) операций чтения/запись

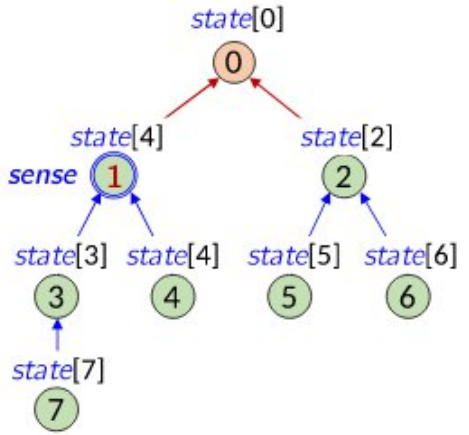
- R. Graham, G. Shipman. ***MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives //*** 2008, DOI: 10.1007/978-3-540-87475-1_21
- S. Jain et al. ***Framework for Scalable Intra-Node Collective Operations using Shared Memory***" // 2018, DOI: 10.1109/SC.2018.00032

Алгоритмы на базе разделяемой памяти

Алгоритм объединяющего дерева (Combining Tree Barrier, CT) – процессы организованы в завершенное k-арное дерево

- **Arrival**: внутренние узлы ожидают дочерние: *state*[*rank*] = *state*[*child*]
- **Departure**: корень устанавливает *sense* = *sense_local*[*root*]

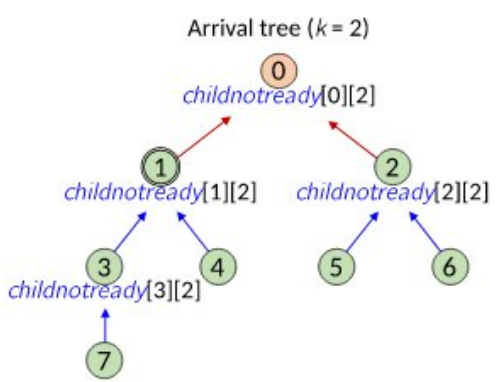
Время: $O(k\log(p))$ операций чтения/запись

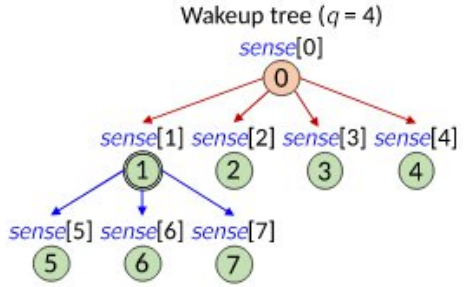


MCS-барьер – каждый процесс входит в два дерева: завершенное k-арное *дерево прибытия* (arrival tree), завершенное q-арное *дерево «пробуждения»* (wakeup tree)

- **Arrival**: внутренние процессы ждут дочерние: *childnotready*[*rank*][*child*] = 1
- **Departure**: процесс ждет уведомления от родителя: *sense*[*parent*][*rank*] = *sense_local*[*rank*]

Время: $O((k + q)\log(p))$ операций чтения/записи operations

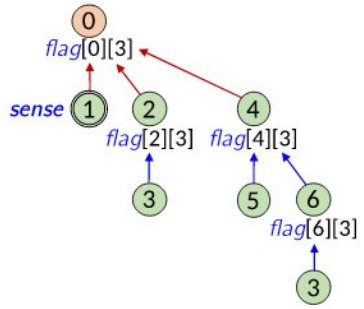




- P.-C. Yew, N.-F. Tzeng and Lawri. ***Distributing Hot-Spot Addressing in Large-Scale Multiprocessors*** // 1987, DOI: 10.1109/TC.1987.1676921
- J. Mellor-Crummey, M. Scott. ***Algorithms for Scalable Synchronization on Shared-memory Multiprocessors*** // 1991, DOI: 10.1145/103727.103729

Алгоритмы на базе разделяемой памяти

Турнирный алгоритм (Tournament Tree Barrier, TR) –
процессы организованы в биномиальное дерево



- **Arrival**: на шаге $k = 0, 1, \ldots, \lceil \log_2 p \rceil - 1$ процесс взаимодействует с процессом $peer = rank \text{ XOR } 2^k$, *проигравший* ($rank > peer$) устанавливает $flag[peer][k]$
- **Departure**: корень 0 устанавливает $sense = sense\_local[root]$

Время: $O(\log(p))$

Рассеивающий алгоритм (Dissemination Barrier, DS) –
децентрализованный алгоритм

- На шаге k процесс ожидает оппонента $peer$: $state[peer] = state[rank]$, $peer = (rank + 2^k) \% p$
- Время: O(log(p))

- D. Hengsen, R. Finkel and U. Manber. ***Two Algorithms for Barrier Synchronization*** // 1988, DOI: 10.1007/BF01379320
- E. Brooks. ***The Butterfly Barrier*** // 1986, DOI: 10.1007/BF01407877

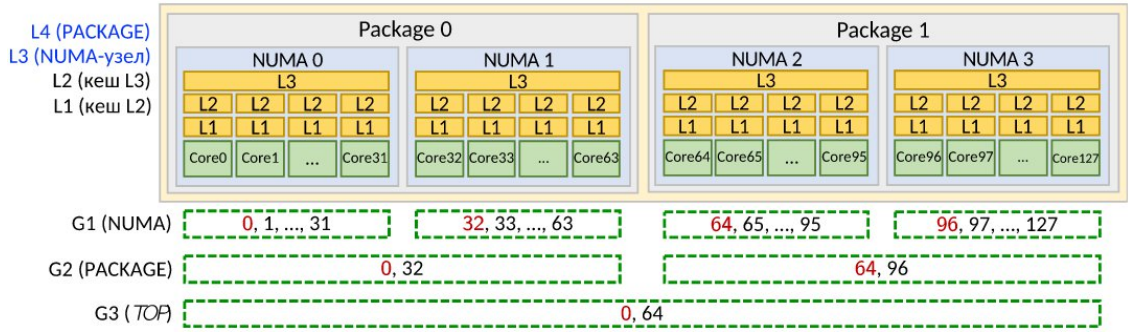
Иерархические алгоритмы барьерной синхронизации

```
algorithm MPI_Barrier()
  sense_local[rank] = sense_local[rank] xor 1
  BarrierGroupLevel(0)
end algorithm

algorithm BarrierGroupLevel(level)
  grank = group[level].group_rank         // Индекс процесса в группе
  gleader = group[level].group_rank_leader  // Индекс лидера в группе
  group[level].state[grank]++             // Уведомление лидера о входе

  if grank = gleader then
    while true do       // Лидер ожидает уведомления от процессов группы
      narrived = 0
      for child = 0 to group[level].size - 1 do
        if group[level].state[child] >= group[level].state[gleader] then
          narrived++
      end for
      if narrived = group[level].size then break
    end while

    if level + 1 < ngroups then     // Лидер переходит на следующий уровень
      BarrierGroupLevel(level + 1)
    else if rank = 0 then
      sense = sense_local[rank]     // Лидер уведомляет о выходе из барьера
    end if
  else
    while sense != sense_local[rank] do  // Ожидание уведомления от лидера 0
  end if
end algorithm
```



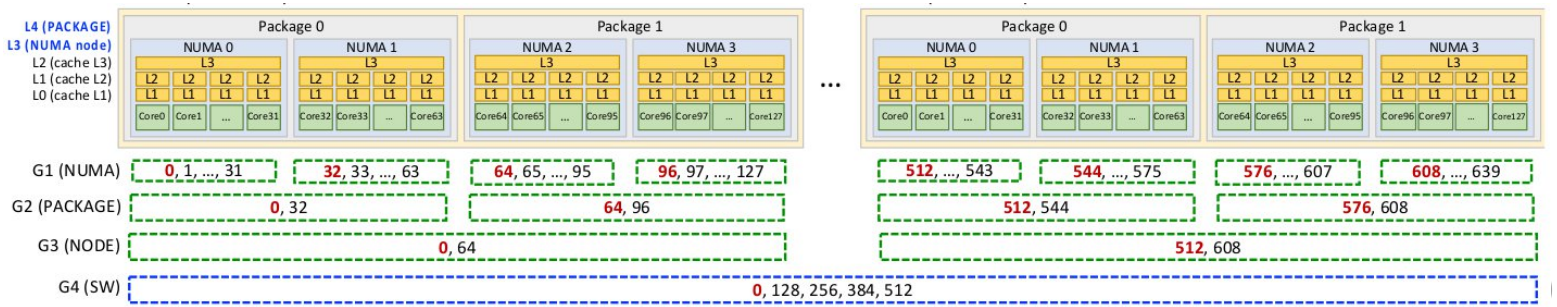
Курносов М.Г. ***Иерархический алгоритм барьерной синхронизации для многопроцессорных систем с общей памятью*** // Вестник СибГТУИ. – №2. – 2022.

Иерархический алгоритм Allreduce

```
function Allreduce(sbuf, rbuf, count, dtype, op, comm)
    if MPI_COMM_NULL != node_comm then
        // Intra-node reduce to local leader
        if MPI_IN_PLACE == sbuf && rank(node_comm) != 0) then
            Reduce(rbuf, NULL, count, dtype, op, 0, node_comm)
        else
            Reduce(sbuf, rbuf, count, dtype, op, 0, node_comm)
        end if
    else
        // One process on a node
        if MPI_IN_PLACE != sbuf then
            // Copy sbuf to rbuf
            Copy(dtype, count, rbuf, sbuf)
        end if
    end if
    // Inter-node Allreduce
    if MPI_COMM_NULL != masters_comm then
      Allreduce(MPI_IN_PLACE, rbuf, count, dtype, op, masters_comm)
    end if
    // Intra-node broadcast among local processes
    if MPI_COMM_NULL != node_comm then
        Bcast(rbuf, count, dtype, 0, node_comm)
    end if
end function
```

Allreduce (two level)

1. Intra-node Reduce to local leader
2. Allreduce on the partial results among all leaders
3. Broadcast the final result within the local node



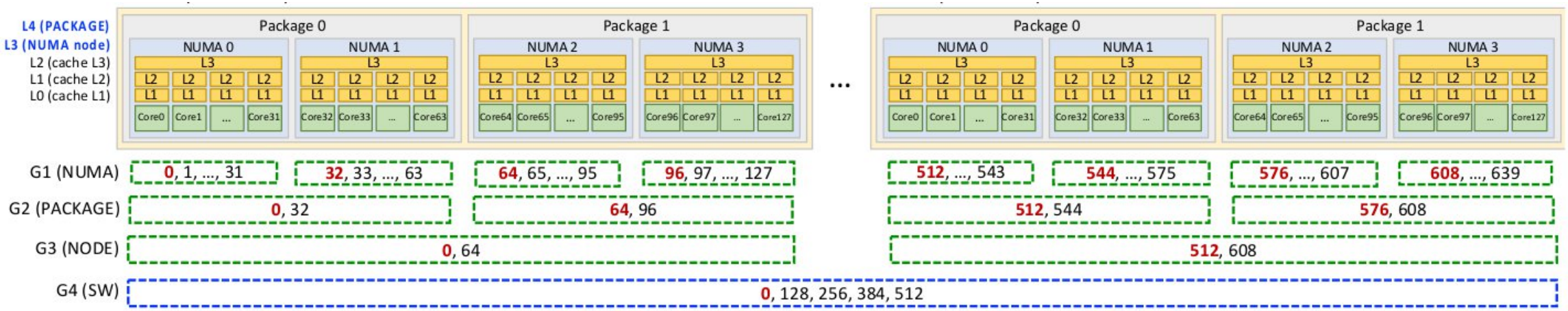
Иерархический алгоритм Allreduce

```
// Создание коммуникаторов intra-node и inter-node

MPI_Comm node_comm = MPI_COMM_NULL;
MPI_Comm leaders_comm = MPI_COMM_NULL;

MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0, MPI_INFO_NULL, &node_comm);
MPI_Comm_rank(node_comm, &node_comm_rank);
int is_node_leader = (node_comm_rank == 0) ? 1 : 0;

MPI_Comm_split(MPI_COMM_WORLD, is_node_leader, 0, &leaders_comm);
```



Иерархический алгоритм Allreduce

```
int MPI_Allreduce_hier(const void *sbuf, void *rbuf, int count, MPI_Datatype dtype, MPI_Op op,
                       MPI_Comm comm, MPI_Comm node_comm, MPI_Comm leaders_comm)
{
    int err = MPI_SUCCESS;
    int node_comm_size, node_comm_rank;
    MPI_Comm_rank(node_comm, &node_comm_rank);
    MPI_Comm_size(node_comm, &node_comm_size);

    /* Intra-node reduce to a local leader */
    if ((MPI_IN_PLACE == sbuf) && (0 != node_comm_rank))  {
        err = MPI_Reduce(rbuf, NULL, count, dtype, op, 0, node_comm);
    } else {
        err = MPI_Reduce(sbuf, rbuf, count, dtype, op, 0, node_comm);
    }
    if (MPI_SUCCESS != err) { goto cleanup_and_return; }

    /* Inter-node allreduce */
    if (MPI_COMM_NULL != leaders_comm) {
        err = MPI_Allreduce(MPI_IN_PLACE, rbuf, count, dtype, op, leaders_comm);
        if (MPI_SUCCESS != err) { goto cleanup_and_return; }
    }
    /* Intra-node broadcast among local processes */
    if (node_comm_size > 1) {
        err = MPI_Bcast(rbuf, count, dtype, 0, node_comm);
    }
cleanup_and_return:
    return err;
}
```