Параллельное программирование (параллельные алгоритмы)

Лисицын Сергей
ФРКТ МФТИ 2018 г.

Программа 1 семестра

513 группа

Занятия: 07.02, 21.02, 7.03, 21.03, 04.04, 18.04, 2.05

Зачёт: 16.05

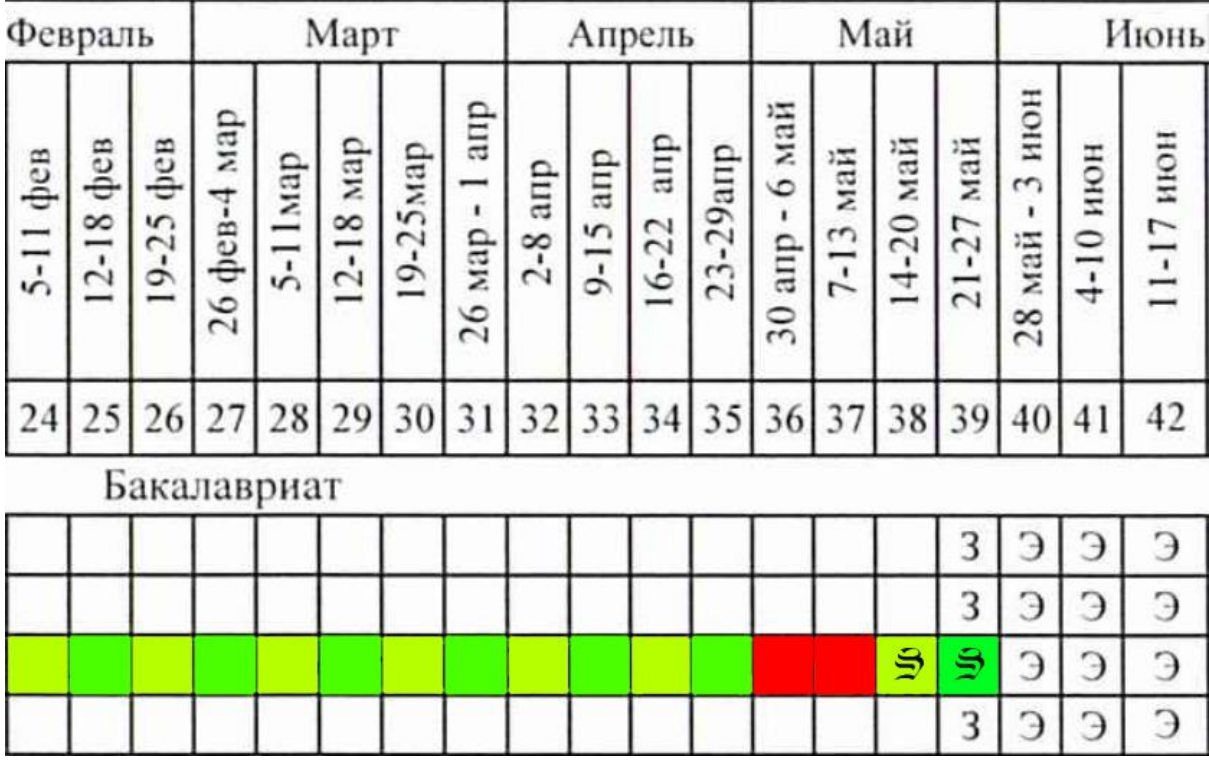
516 группа

Занятия: 14.02, 28.02, 14.03, 28.03, 11.04, 25.04, 9.05

Зачёт: 23.05

Среда										
	9^{00} - 10^{25}	Квантовая механика/ Доцент Гец А.В./ Б.Физ.							Операционные системы/ 802 КПМ	
	10^{45} - 12^{10}	Выч.математика 705 КПМ		Квант.механика 514 ГК	Физкультура	Физкультура	Физкультура		Операционные системы/ 702 КПМ	
	12^{20} - 13^{45}		Выч.математика 705 КПМ					Квант.механика 514 ГК		
	13^{55} - 15^{20}	Вычислительная математика/ Доцент Барабанщиков А.В./ Актовый зал								
	15^{30} - 16^{55}	Ин.яз.	Ин.яз.	Ин.яз.	Ин.яз.	Ин.яз.	Ин.яз.	Ин.яз.	Ин.яз.	Ин.яз.
	17^{05} - 18^{30}	Параллельное программирование/ Чл.-корр. РАН Якобовский М.В./ Б.Хим.								
	18^{35} - 20^{00}			Паралл.програм. (неч.нед.) 801 КПМ			Паралл.програм. (чет.нед.) 801 КПМ		Паралл.прогр. (неч.нед.)804 КПМ	Паралл.програм. (чет.нед.) 804 КПМ

Программа 1 семестра



6 семинаров*:

1. Вводная лекция
2. Программирование на MPI
3. Статическая балансировка
4. Динамическая балансировка
5. Стандарт POSIX Threads
6. Программирование на общей памяти

*план может быть изменён

Программа 1 семестра

Программы:

- Обязательные задачи (4)
- Бонусные задачи (3)

Оценка за семестр:

+5: Лекционная контрольная

+2: Посещения/3

+2: Мгновенные обязательные задачи/2

+3: Бонусные задачи

Введение

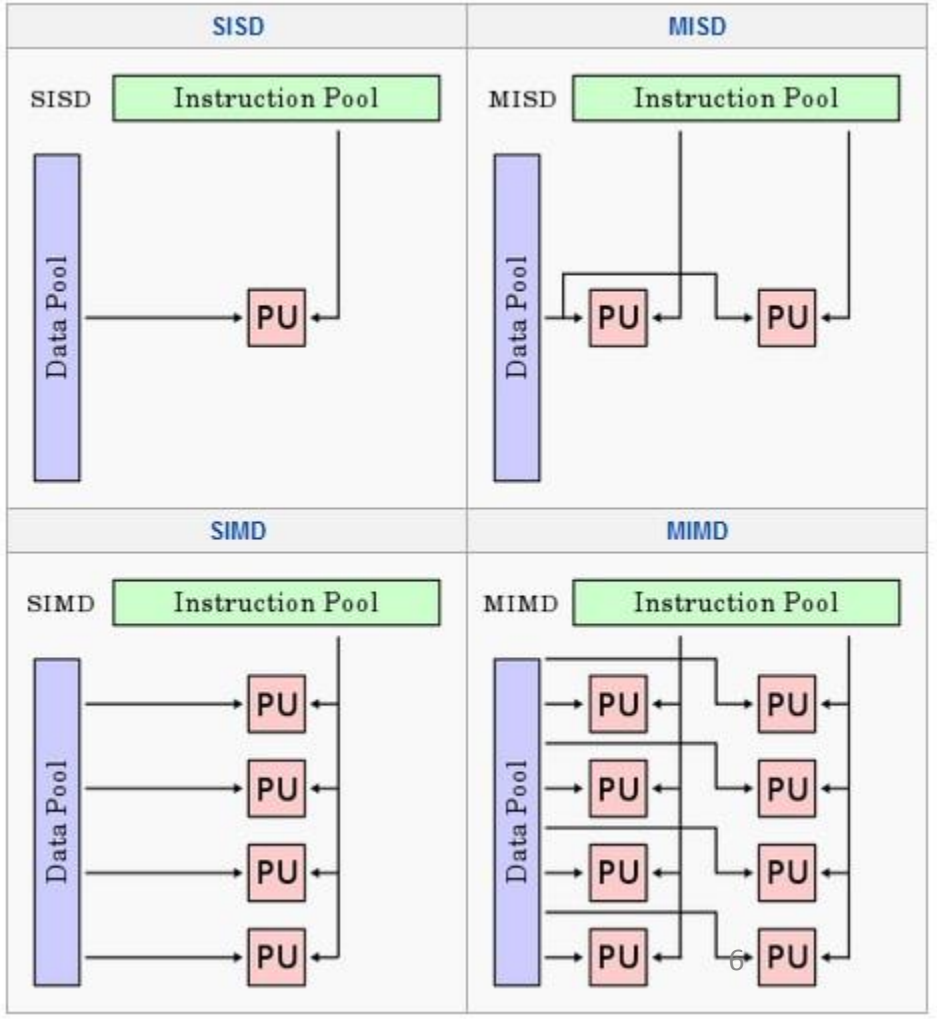
Perf = Freq*IPC/IC

- Конвейеризация вычислений (1970-е) – микроуровневый параллелизм
- Дублирование вычислителей (1980-е) – параллелизм уровня команд (векторизация, VLIW)
- Дублирование “конвейеров” (2000-е)- параллелизм уровня потоков/заданий

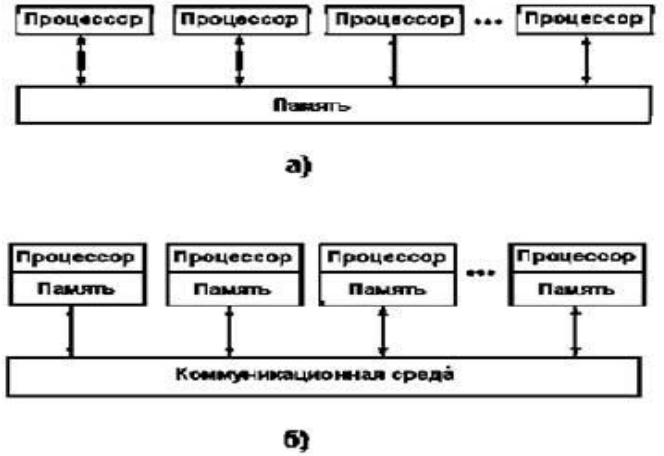
Введение

Таксономия (Классификация) Флинна (1966)

- SISD: компьютер фон-Неймановской архитектуры
- SIMD: векторные процессоры (MMX, SSE), матричные процессоры и процессоры с архитектурой VLIW.
- MISD: не используется
- MIMD:
  - Общая память - Symmetric Multiprocessor SMP
    Разделенная память - Massively Parallel Processing MPP -> Кластерные системы



Введение



Симметричное мультипроцессирование

1. Несколько однородных процессоров и массив общей памяти
2. Когерентность кэшей, урегулирование доступа к памяти
3. Ограниченная масштабируемость
4. Работает под единой ОС
5. Модель программирования: Потоки (pthread, OpenMP)

Массивно-параллельные системы

1. Вычислительные узлы и коммуникационная среда
2. Закрытая локальная память
3. Масштабируемость ~не ограниченная
4. Полная ОС на управляющей машине, на узлах урезанная версия
5. Модель программирования: Модель передачи сообщений (“fork”, MPI, PVM, BSPlib)

Введение

Метрики параллелизма

- Доля последовательных операций: $\alpha = \frac{IC_{|}}{(IC_{|} + IC_{||})}$
- Время на p потоках: $T_p = \alpha T_1 + \frac{(1-\alpha)T_1}{p}$
- Ускорение: $S = T_1/T_p$
- Эффективность: $E = S/p$

Закон Амдала

$$S = \frac{T_1}{T_p} = \frac{T_1}{\alpha T_1 + \frac{(1-\alpha)T_1}{p}} \leq \frac{1}{\alpha}$$

MPI

Message Passing Interface

Параллельная программа - множество одновременно выполняемых **процессов.**

Каждый процесс порождается на основе одного и того же программного кода (fork).

Количество процессов определяется в момент запуска программы.

Все процессы последовательно пронумерованы от 0 до **p-1** (ранг процесса), где **p** есть общее количество процессов.

MPI

- Подключение библиотеки:

```
#include "mpi.h"
```

- Начало работы MPI:

```
int MPI_Init ( int *agrc,
char ***argv );
```

- Завершение работы:

```
MPI_Finalize();
```

```
#include "mpi.h"

int main ( int argc, char *argv[] )
{
        <программный код без MPI функций>
    MPI_Init ( &agrc, &argv );
        <программный код с MPI функциями>
    MPI_Finalize();
        <программный код без MPI функций>
    return 0;
}
```

Функции **MPI_Init** и **MPI_Finalize** являются обязательными и должны быть выполнены (только один раз) каждым процессом параллельной программы.

MPI

Компиляция программы:

mpicc -o <исполняемый файл> <исходный файл>.c

Запуск:

mpirun -n <число процессов>

<исполняемый файл> [аргументы]

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
    int ProcRank;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    printf("Hello from process %d\n", ProcRank);

    MPI_Finalize();
    return 0;
}

/// mpicc -o hello main.c
/// mpirun -n 4 hello
```

MPI

- Определение количества процессов

```
int MPI_Comm_size ( MPI_Comm comm, int *size );
```

- Определения ранга процесса

```
int MPI_Comm_rank ( MPI_Comm comm, int *rank )
```

```
#include "mpi.h"

int main ( int argc, char *argv[] )
{
    int ProcNum, ProcRank;
        <программный код без использования MPI функций>
    MPI_Init ( &agrc, &argv );
    MPI_Comm_size ( MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank ( MPI_COMM_WORLD, &ProcRank);
        <программный код с использованием MPI функций>
    MPI_Finalize();
        <программный код без использования MPI функций>
    return 0;
}
```

MPI

Коммуникатор - служебный объект, объединяющий в своем составе группу процессов и контекст, используемый при передачи данных.

Все процессы входят в состав создаваемого по умолчанию коммуникатора с идентификатором MPI_COMM_WORLD.

MPI_Comm comm = MPI_COMM_WORLD;

MPI

Операции передачи сообщений:

- Парные (point-to-point) – операции между двумя процессами

  MPI_Send, MPI_Recv…

- Коллективные (collective) – коммуникационные действия для **одновременного** взаимодействия нескольких процессов

  MPI_Bcast, MPI_Reduce…

MPI

Базовые типы

MPI_Datatype	C Datatype
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	Double
MPI_FLOAT	Float
MPI_INT	Int
MPI_LONG	Long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Производные типы

- Функции создания типа
  MPI_Type_contiguous
  MPI_Type_vector
  MPI_Type_hvector
  MPI_Type_indexed
  MPI_Type_hindexed
  MPI_Type_struct

- Функция регистрации типа
  `int` **`MPI_Type_commit`**
  `(MPI_Datatype *datatype);`

MPI

Передача сообщений:

```
int MPI_Send(void *buf, int count,
MPI_Datatype type, int dest, int tag,
MPI_Comm comm);
```

Приём сообщений:

```
int MPI_Recv(void *buf, int count,
MPI_Datatype type, int source, int tag,
MPI_Comm comm, MPI_Status *status);
```

MPI_ANY_SOURCE

MPI_ANY_TAG

MPI_COMM_WORLD

buf – адрес буфера с данными отправляемого сообщения.
count – количество элементов данных в сообщении.
type - тип элементов данных пересылаемого сообщения.
dest - ранг процесса, которому отправляется сообщение.
source - ранг процесса, от которого должен быть выполнен прием сообщения.
tag - значение-тег, используемое для идентификации сообщений.
comm - коммуникатор, в рамках которого выполняется передача данных.
status – указатель на структуру данных с информацией о результате выполнения операции.

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char* argv[])
{
    int ProcNum, ProcRank, RecvRank;
    MPI_Status Status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);

    if ( ProcRank == 0 )
    {
        /// Действия, выполняемые только процессом с рангом 0
        printf ("\n Hello from process %d", ProcRank);
        for ( int i = 1; i < ProcNum; i++ )
        {
            MPI_Recv(&RecvRank, 1, MPI_INT, MPI_ANY_SOURCE,
            MPI_ANY_TAG, MPI_COMM_WORLD, &Status);
            printf("\n Hello from process %d", RecvRank);
        }
    } else {
        /// Сообщение, отправляемое всеми процессами,
        /// кроме процесса с рангом 0
        MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

MPI

Описание всех функций:
https://www.open-mpi.org/doc/current/

Подробное описание MPI на русском:
http://rsusu1.rnd.runnet.ru/tutor/method/m2/content.html

MPI

Измерение времени:

```
double MPI_Wtime(void);
```

Измерение точности измерения времени:

```
double MPI_Wtick(void);
```

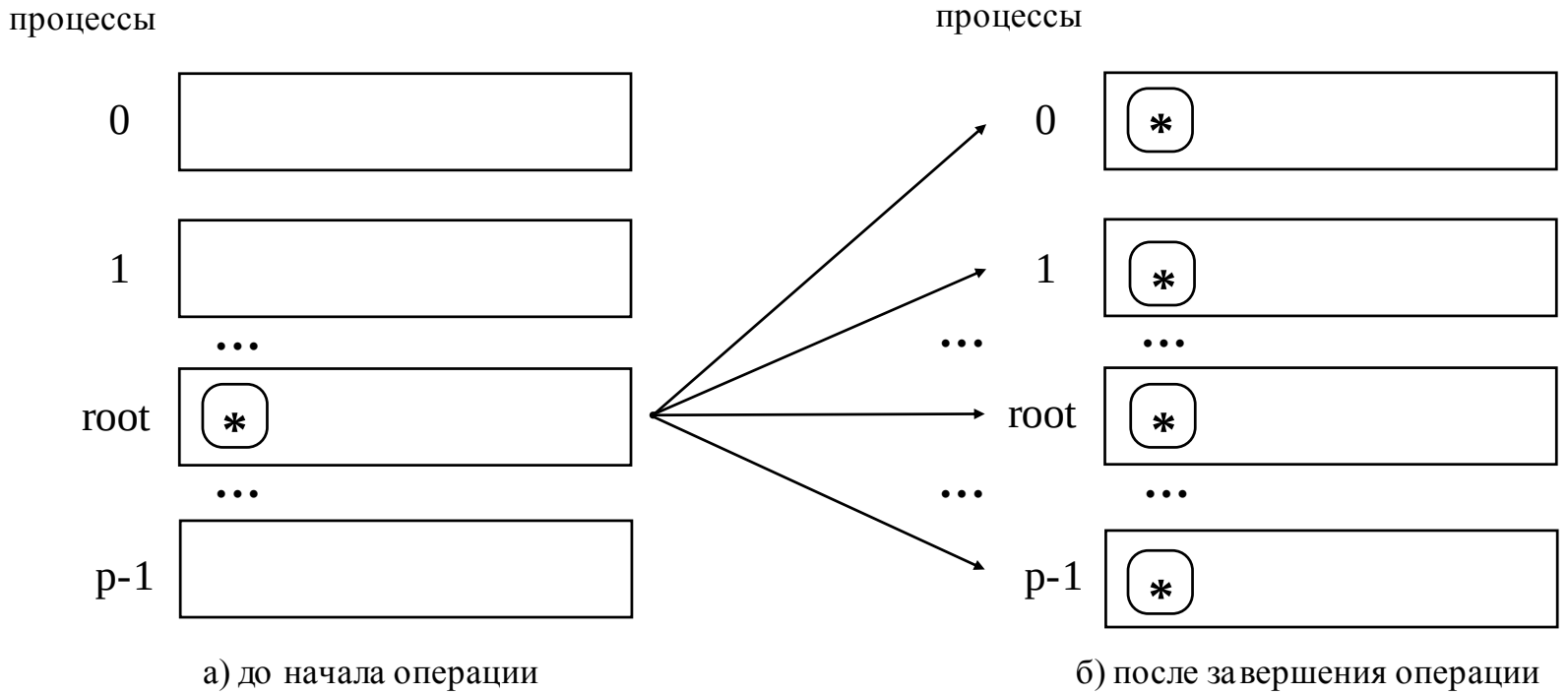
Синхронизация процессов:

```
int MPI_Barrier(MPI_Comm comm);
```

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv)
{
  int rank = 0, size = 0;
  double start_time = 0, end_time = 0;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  MPI_Barrier(MPI_COMM_WORLD);
  if (rank == 0)
    start_time = MPI_Wtime();

  /// Алгоритм,врямя которого измеряем

  MPI_Barrier(MPI_COMM_WORLD);
  if (rank == 0)
    end_time = MPI_Wtime();

  if (rank == 0)
    printf("[TIME] %lf\n", end_time - start_time);

  MPI_Finalize();
  return 0;
}
```

MPI

Широковещательная рассылка данных




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

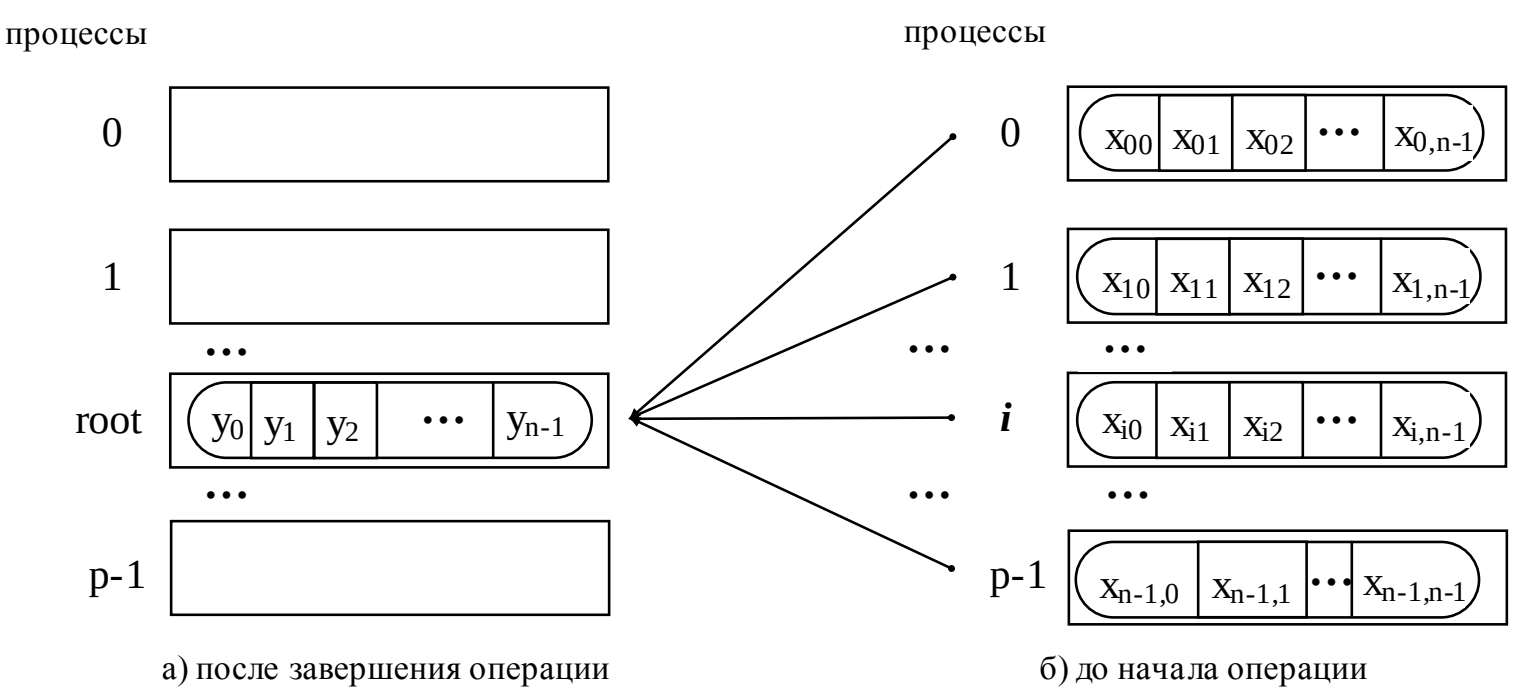
buf – буфер памяти с отправляемым сообщением (для процесса с рангом 0), и для приема сообщений для всех остальных процессов

root – ранг процесса, выполняющего рассылку данных

MPI

Сбор данных с операцией

$$y_j = \bigotimes_{i=0}^{n-1} x_{ij}, 0 \le j < n$$




```
int MPI_Reduce(void *sendbuf, void *recvbuf,int
count, MPI_Datatype type, MPI_Op op,int root,MPI_Comm
comm)
```

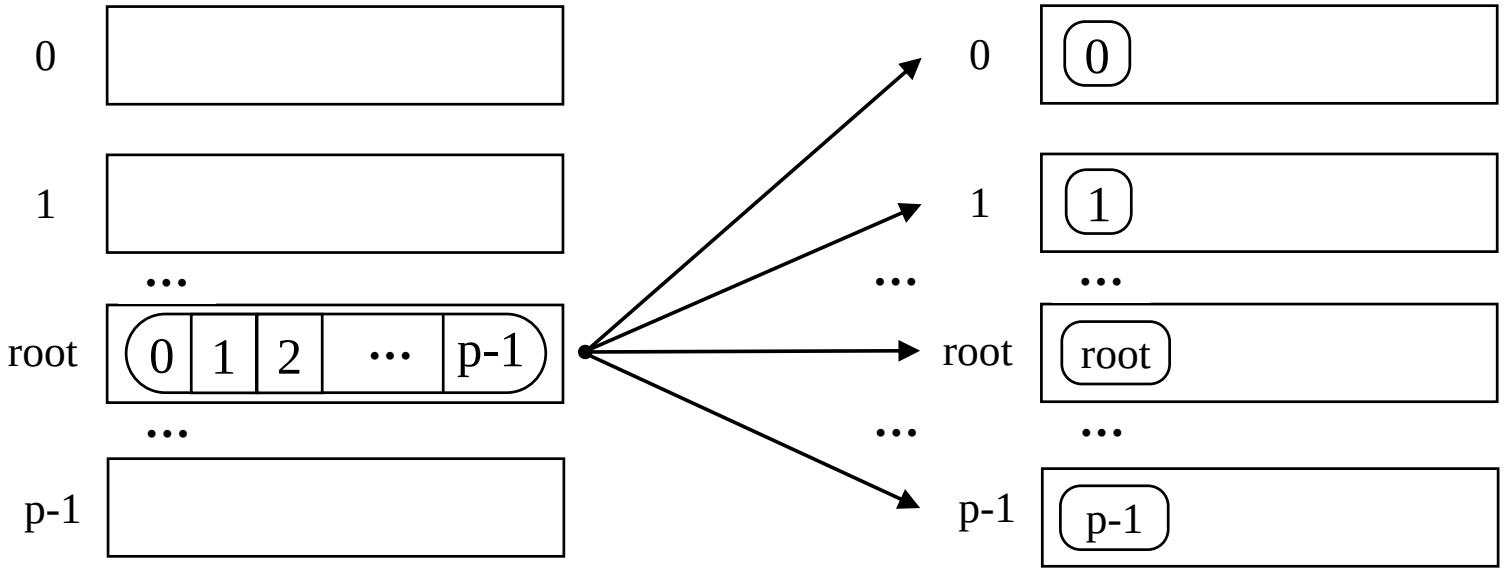
- **sendbuf** – буфер памяти с отправляемым сообщением
- **recvbuf** – буфер памяти для результирующего сообщения (только для процесса с рангом `root`)
- **op** – операция, которая должна быть выполнена над данными
- **root** – ранг процесса, на котором должен быть получен результат

MPI

Операция	**Описание**
MPI_MAX	Определение максимального значения
MPI_MIN	Определение минимального значения
MPI_SUM	Определение суммы значений
MPI_PROD	Определение произведения значений
MPI_LAND	Выполнение логической операции "И" над значениями сообщений
MPI_BAND	Выполнение битовой операции "И" над значениями сообщений
MPI_LOR	Выполнение логической операции "ИЛИ" над значениями сообщений
MPI_BOR	Выполнение битовой операции "ИЛИ" над значениями сообщений
MPI_LXOR	Выполнение логической операции исключающего "ИЛИ" над значениями сообщений
MPI_BXOR	Выполнение битовой операции исключающего "ИЛИ" над значениями сообщений
MPI_MAXLOC	Определение максимальных значений и их индексов
MPI_MINLOC	Определение минимальных значений и их индексов

MPI

Обобщенная передача данных от одного всем



а) до начала операции

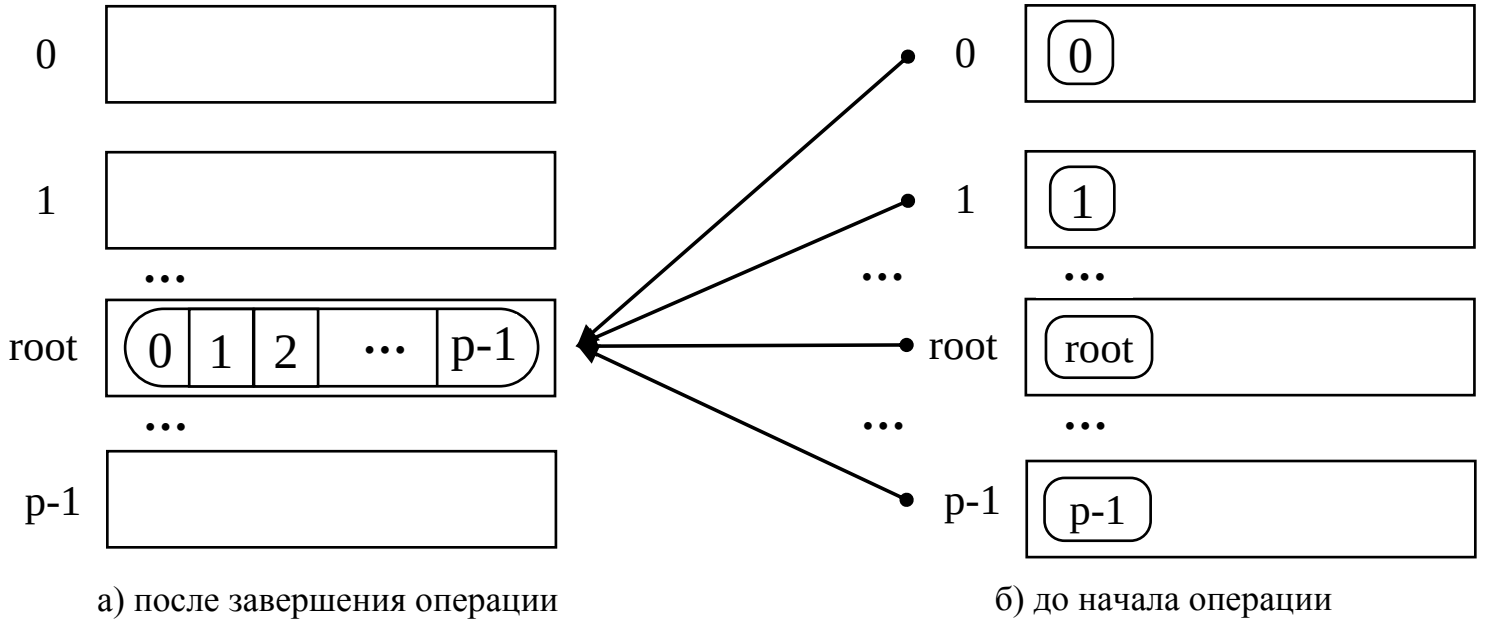
б) после завершения операции

```
int MPI_Scatter(void *sbuf,int scount,MPI_Datatype
stype, void *rbuf,int rcount,MPI_Datatype rtype, int
root, MPI_Comm comm)
```

- **sbuf, scount, stype** – параметры передаваемого сообщения (**scount** определяет количество элементов, передаваемых на каждый процесс)
- **rbuf, rcount, rtype** – параметры сообщения, принимаемого в процессах
- **root** – ранг процесса, выполняющего рассылку данных

MPI

Обобщенная передача данных от всех одному



а) после завершения операции

б) до начала операции

```
int MPI_Gather(void *sbuf,int scount,MPI_Datatype
stype, void *rbuf,int rcount,MPI_Datatype rtype, int
root, MPI_Comm comm)
```

- **sbuf**, **scount**, **stype** - параметры передаваемого сообщения,
- **rbuf**, **rcount**, **rtype** - параметры принимаемого сообщения,
- **root** – ранг процесса, выполняющего сбор данных,

MPI

MPI_All* - результат дублируется во всех процессах (MPI_Allreduce, MPI_Allgather …)

MPI_*v – размеры передаваемых процессами сообщений могут быть различны (MPI_Scatterv, MPI_Allgatherv …)

MPI

Режимы передачи данных:

MPI_Send – стандартный режим.

MPI_Ssend – синхронный режим (ждет ответ о приёме сообщения)

MPI_Rsend – режим передачи по готовности (сработает только при запущенном MPI_Recv)

MPI_Bsend – буферизированный режим (использует дополнительный буфер для отправки)

MPI

Блокирующие функции приостанавливают выполнение процессов до момента завершения работы. (MPI_*Send, MPI_Recv)

Неблокирующие функции обмена данными выполняются без блокировки. (MPI_I*send, MPI_Irecv)

```
int MPI_I*(…, MPI_Request *request);
```

Проверка состояния неблокирующей функции:

```
int MPI_Test( MPI_Request *request, int *flag,
MPI_status *status)
```

-request - дескриптор операции, определенный при вызове неблокирующей функции

-flag - результат проверки (=true, если операция завершена)

-status - результат выполнения операции обмена (только для завершенной операции)

MPI

MPI_Testall – проверка завершения всех перечисленных операций обмена

MPI_Waitall – ожидание завершения всех операций обмена

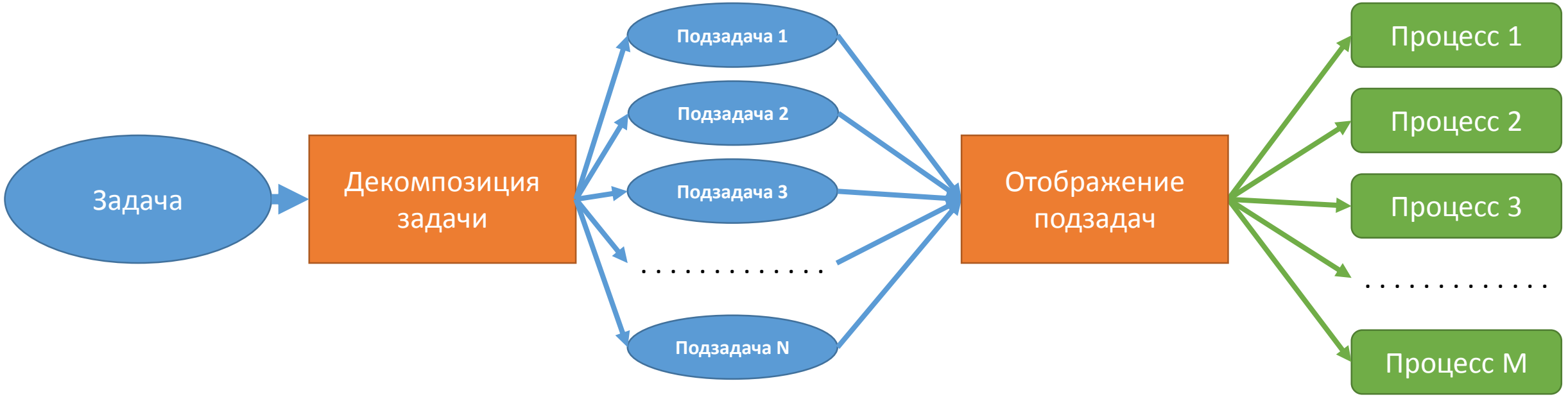
MPI_Testany – проверка завершения хотя бы одной из перечисленных операций обмена

MPI_Waitany – ожидание завершения любой из перечисленных операций обмена

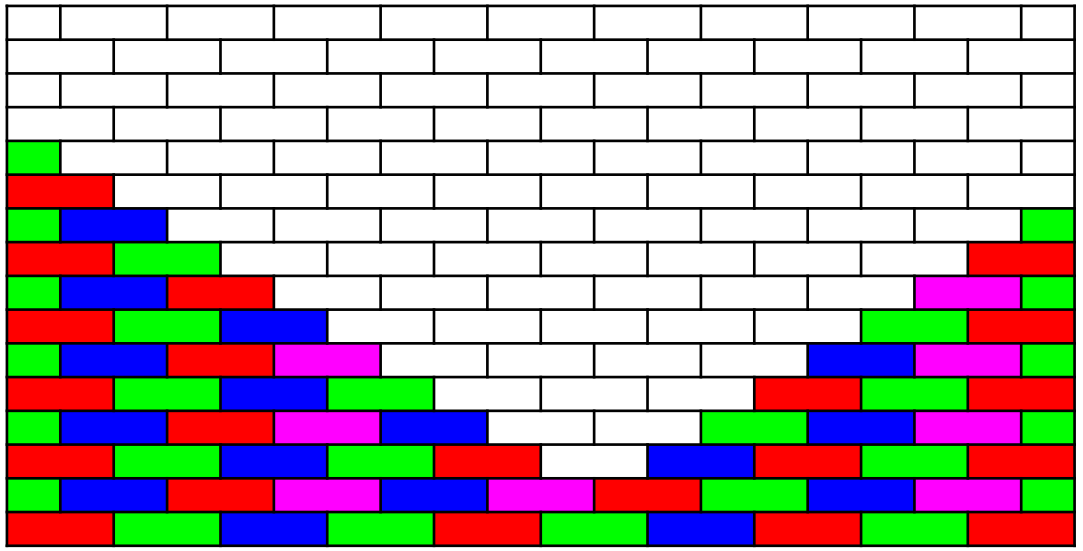
MPI_Testsome – проверка завершения каждой из перечисленных операций обмена

MPI_Waitsome – ожидание завершения хотя бы одной из перечисленных операций обмена и оценка состояния по всем

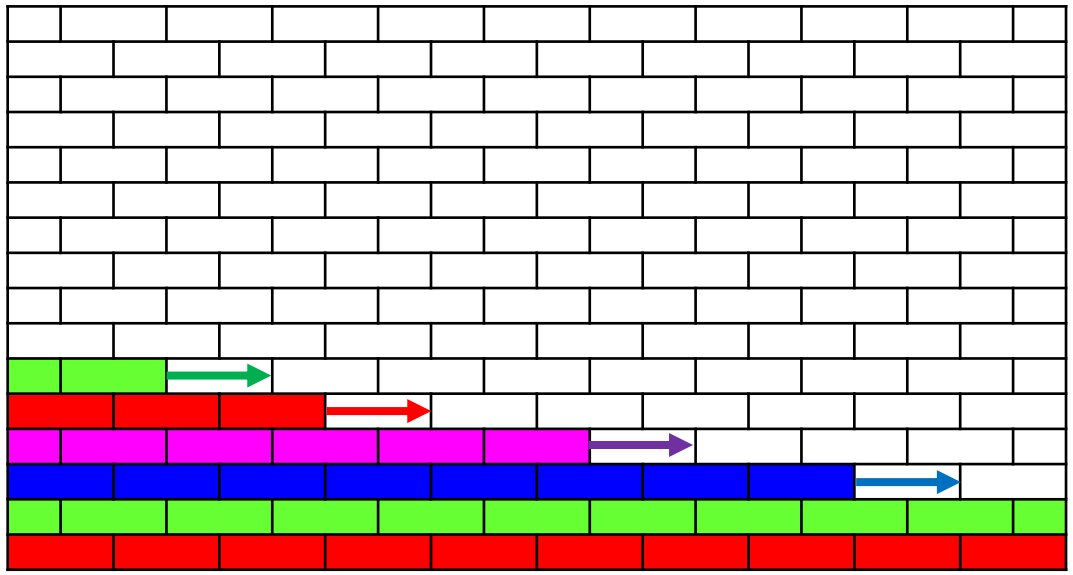
Балансировка нагрузки

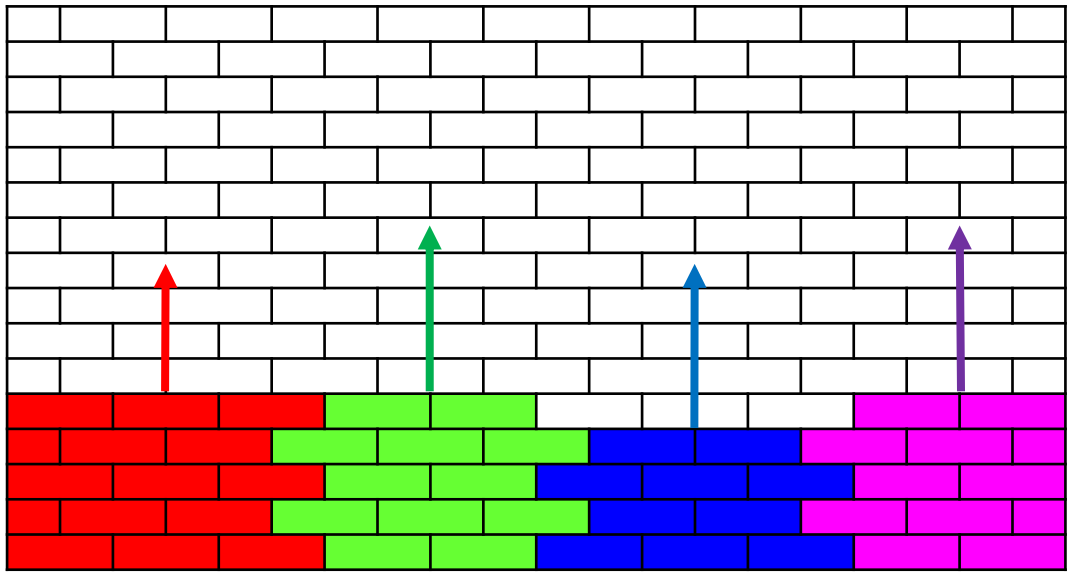


Балансировка нагрузки



Стена Фокса (пример с лекций © М.В. Якобовского)





Балансировка нагрузки

Проблемы балансировки вычислительной нагрузки:

- структура распределенной задачи неоднородна
- структура вычислительного комплекса (например, кластера) неоднородна
- структура межузлового взаимодействия неоднородна

Балансировка нагрузки

Статическая балансировка

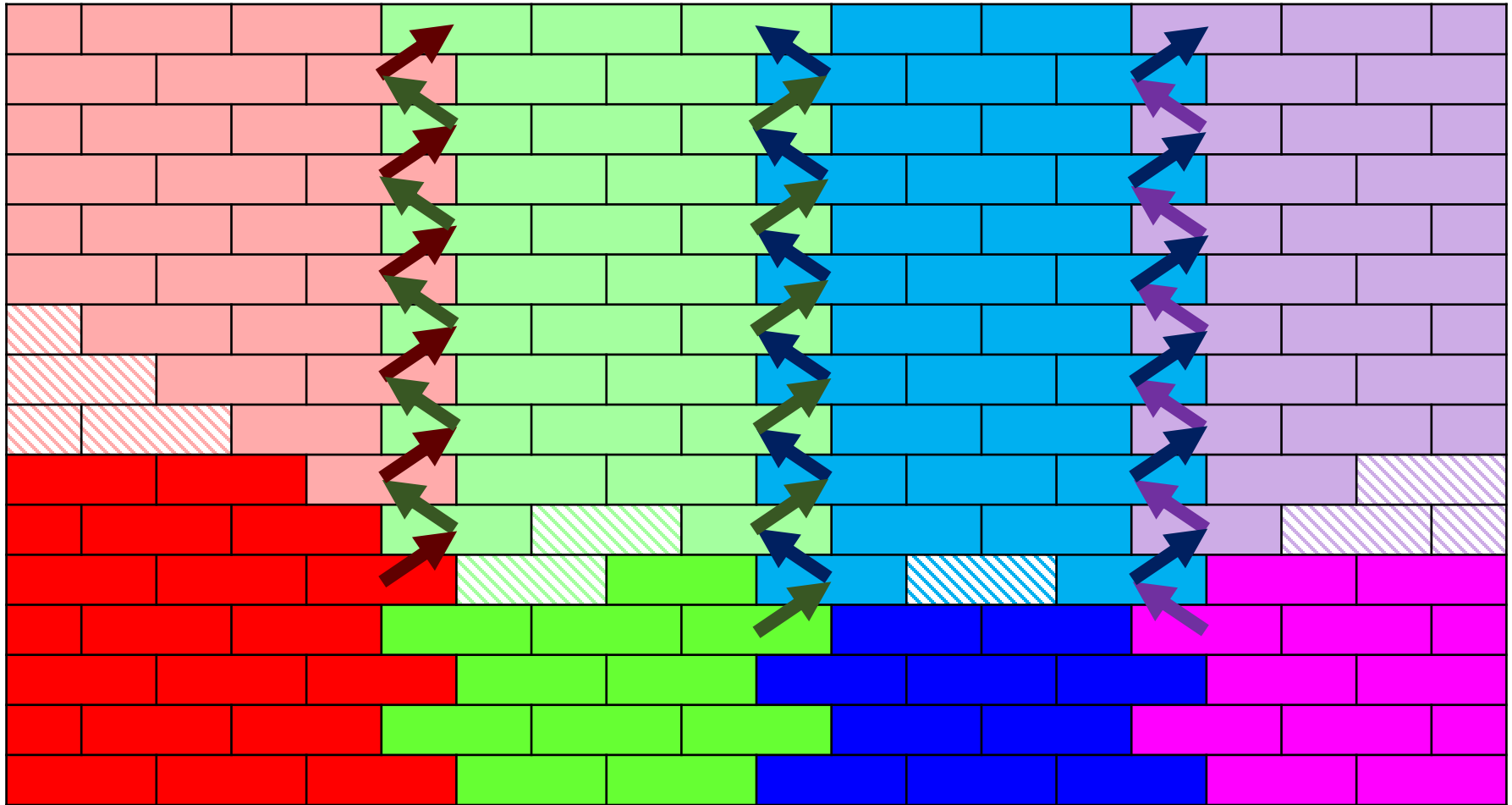
- отображение задач до начала выполнения задачи
- борьба с неоднородностями при помощи эвристик и опыта предыдущих запусков

Динамическая балансировка

- отображение задач происходит до и во время выполнения задачи
- борьба с неоднородностями при помощи постоянного (пере)распределения задач

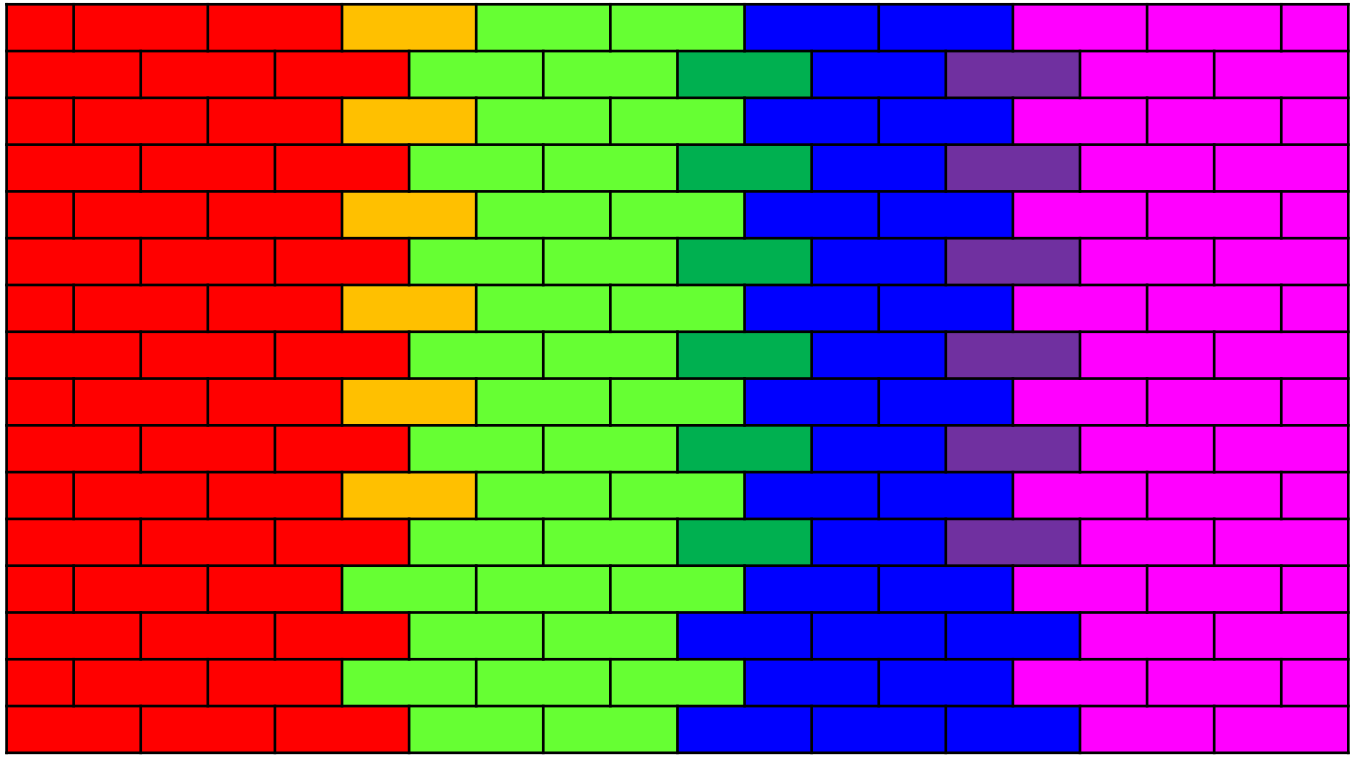
Балансировка нагрузки

Статическая балансировка



Балансировка нагрузки

Динамическая балансировка



Балансировка нагрузки

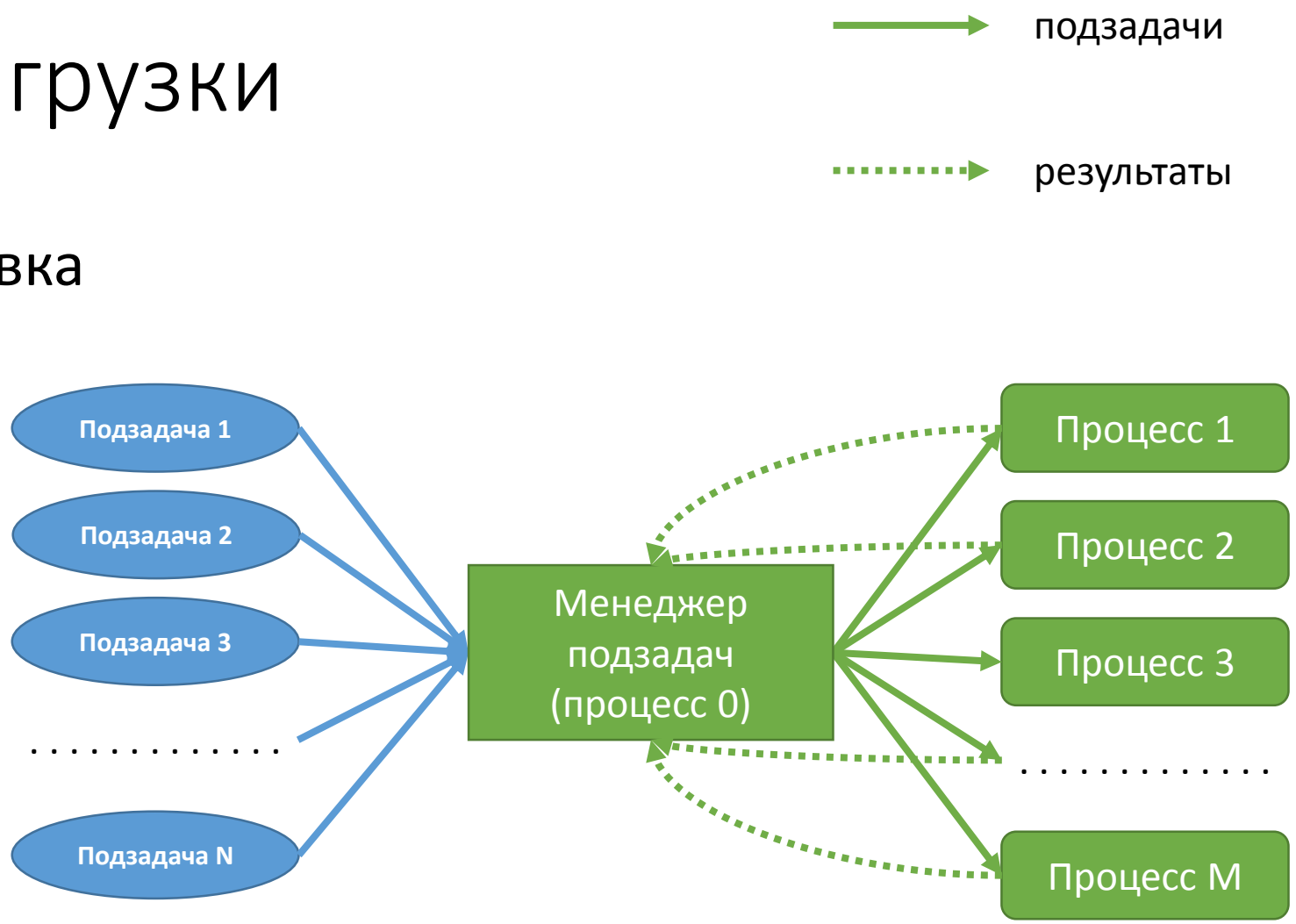
Динамическая балансировка

RCL – стратегия переноса нагрузки:

•случайный алгоритм (random, R)

•алгоритм, основанный на коммуникациях (communication, C)

•алгоритм, основанный на вычислении нагрузки (load, L)



Балансировка нагрузки

Длинная арифметика (сложение)

+	**21**	**43**	**76**	**54**
	4	55	24	02
			0←	56
		1←	100	
	0←	99		
	25	99	00	56

Балансировка нагрузки

Спекулятивные вычисления

	21	**43**	**76**	**54**	**53**	**09**	**12**	**94**	**11**	**23**	**08**	**05**
+	4	55	24	45	85	75	25	41	54	25	08	97
			0←	99			1←	135			1←	102
			1←	100			1←	136			1←	102
		1←	100			0←	38			0←	17	
		1←	101			0←	38			0←	17	
	0←	99			0←	84			0←	48		
	0←	99			0←	84			0←	48		
	25				138				65			
	25				138				65			
	25	99	00	99	38	84	38	35	65	48	17	02
	25	99	01	00	38	84	38	36	65	48	17	02

Балансировка нагрузки

Суммирование старших разрядов

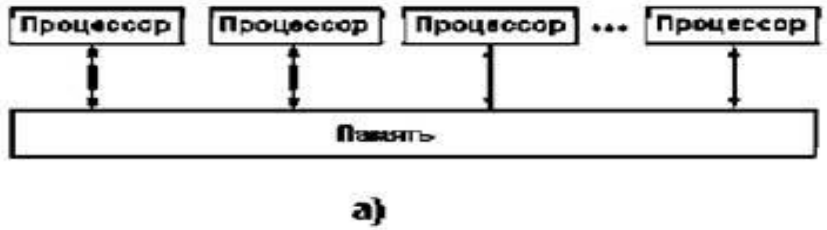
	21	43	76	54	53	09	12	94	11	23	08	05
+	4	55	24	45	85	75	25	41	54	25	08	97
			0←	99	←138		1←	135	←65		1←	102
			1←	100	137		1←	136	66		1←	102
		1←	100			0←	38			0←	17	
		1←	101			0←	38			0←	17	
	0←	99			0←	84			0←	48		
	0←	99			0←	84			0←	48		
	25				138				65			
	25				138				65			
	25	99	00	99	38	84	38	35	65	48	17	02
	25	99	01	00	38	84	38	36	65	48	17	02

Балансировка нагрузки

Динамическая балансировка

Исполнитель					1	3	3 +	2	1 +
Первое число	32 разряда	32 разряда	32 разряда	32 разряда	32 разряда	32 разряда	32 разряда	32 разряда	24 разряда
Второе число	32 разряда	32 разряда	32 разряда	32 разряда	32 разряда	32 разряда	32 разряда	32 разряда	24 разряда
Результат (0)							32 разряда		24 разряда
Результат (1)							32 разряда		24 разряда
Перенос разряда (0)							1 разряд		1 разряд
Перенос разряда (1)							1 разряд		1 разряд

POSIX Threads



а)

б)

POSIX – portable operating system interface

Добавление Threads API:

#include <pthread.h>

Компиляция программы:

gcc -o <исполняемый файл> <исходный файл>.c -lpthread

POSIX Threads

Создание потока:

int **pthread_create**(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);

Ожидание завершения потока:

int **pthread_join**(pthread_t thread, void **status);

```c
#include <pthread.h>
#include <stdio.h>

void * thread_func(int id)
{
    printf("[Thread] %d\n", id);
    return;
}

void main(int argc, char **argv)
{
    pthread_t  thread1, thread2;

    pthread_create(&thread1, NULL, (void *(*) (void *)) thread_func, (void *) 1);
    pthread_create(&thread2, NULL, (void *(*) (void *)) thread_func, (void *) 2);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return;
}
```

POSIX Threads + MPI

Компиляция программы:

mpicc -o <исполняемый файл> <исходный файл>.c -lpthread

Запуск:

mpirun -n <число процессов>

<исполняемый файл> [аргументы]

```c
#include <pthread.h>
#include "mpi.h"
#include <stdio.h>

int rank, size;
void * thread_func(int id)
{
    printf("[Process]%d [Thread] %d\n", rank, id);
    return;
}

void main(int argc, char **argv)
{
    pthread_t  thread1, thread2;

    /// Initialize Mpi Environment.
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    pthread_create(&thread1, NULL, (void *(*) (void *)) thread_func, (void *) 1);
    pthread_create(&thread2, NULL, (void *(*) (void *)) thread_func, (void *) 2);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    MPI_Finalize();
    return;
}
```

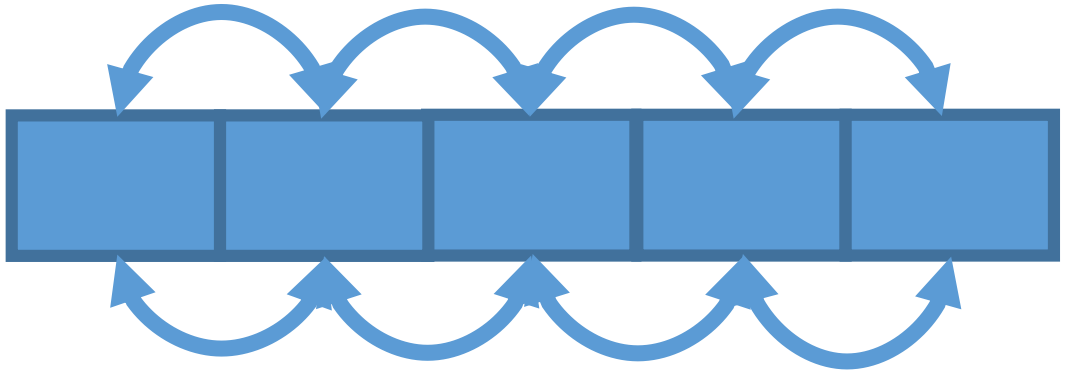
POSIX Threads + MPI

- Потоки довольно просто обмениваются данными по сравнению с процессами.
- Создавать потоки для ОС проще и быстрее, чем создавать процессы.

- Необходима потоковая безопасность функций. Для процессов это не нужно.
- Один бажный поток может повредить остальные. Процессы более изолированы друг от друга.
- Потоки конкурируют друг с другом в адресном пространстве за стек и локальное хранилище.

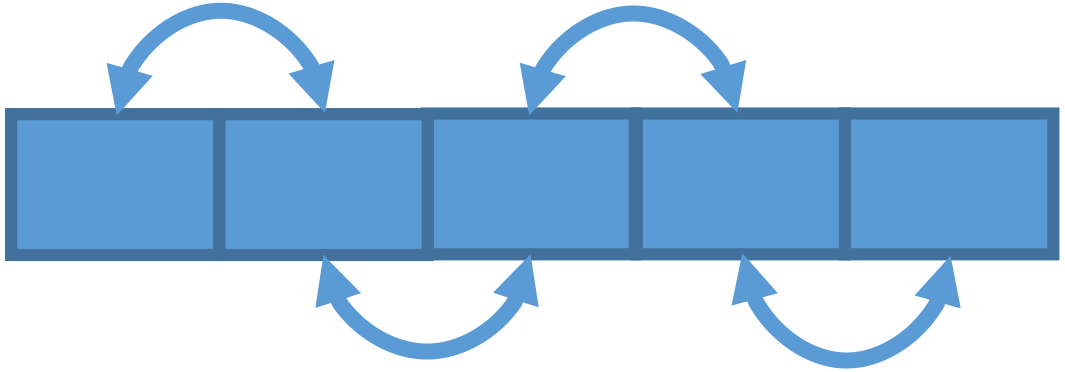
Сортировка

Пузырьковая сортировка



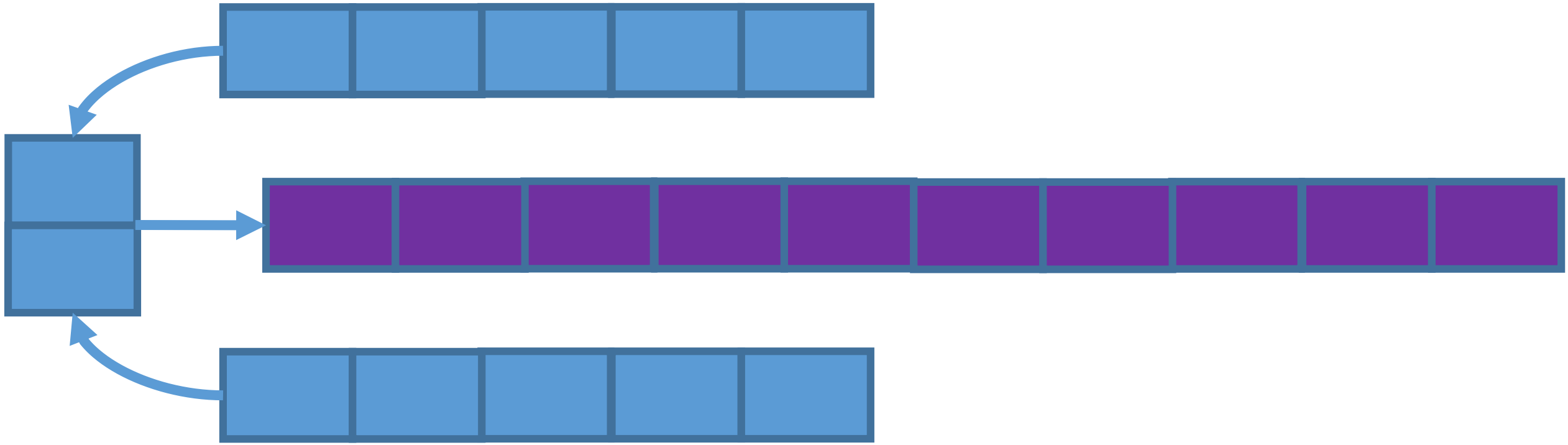
Сортировка

Чёт-нечёт перестановка



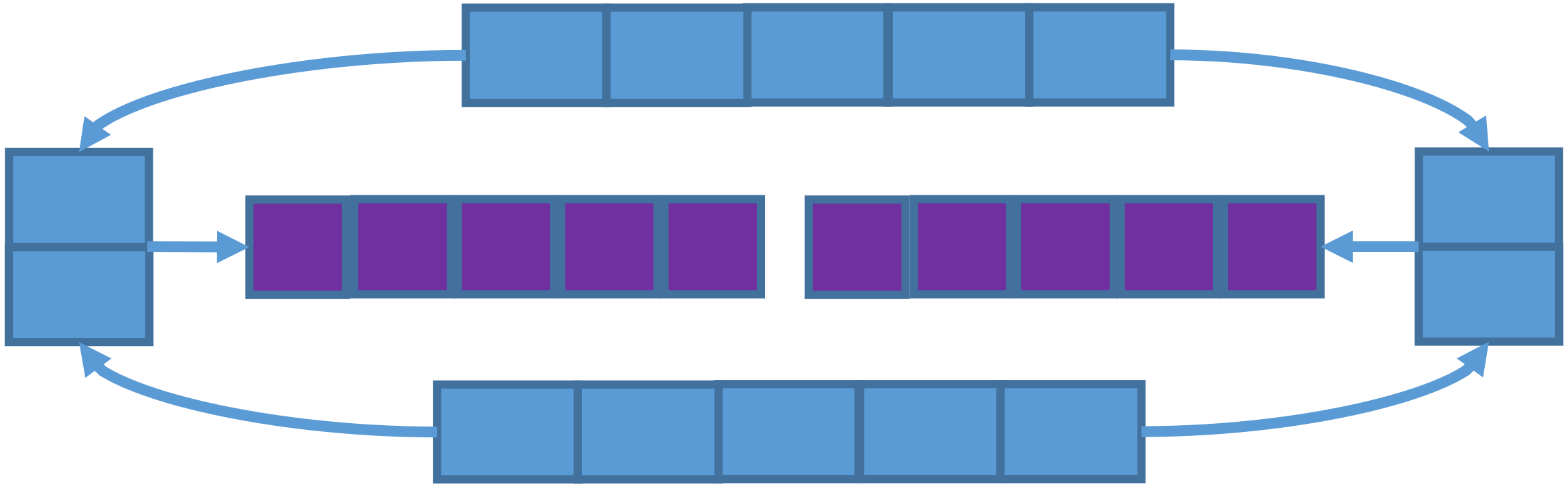
Сортировка

Сортировка слиянием



Сортировка

Сортировка слиянием



Сортировка

Чёт-нечёт слияние