

# **Распределенная общая память (DSM - Distributed Shared Memory)**

# DSM

- Традиционно распределенные вычисления базируются на модели передачи сообщений, в которой данные передаются от процессора к процессору в виде сообщений. Удаленный вызов процедур фактически является той же самой моделью (или очень близкой).
- DSM - виртуальное адресное пространство, разделяемое всеми узлами (процессорами) распределенной системы. Программы получают доступ к данным в DSM примерно так же, как это происходит при реализации виртуальной памяти традиционных ЭВМ. В системах с DSM данные могут перемещаться между локальными памятьями разных компьютеров аналогично тому, как они перемещаются между оперативной и внешней памятью одного компьютера.

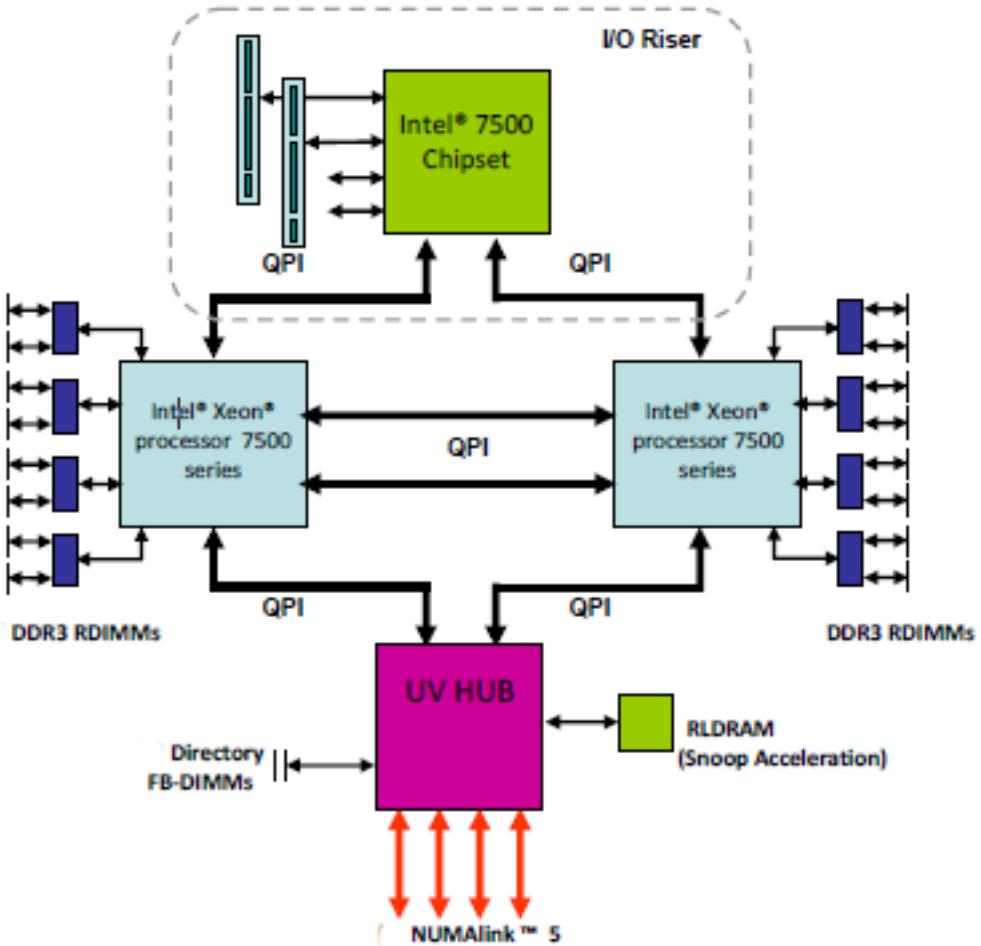
# Достоинства DSM

- 1) В модели передачи сообщений программист обеспечивает доступ к разделяемым данным посредством явных операций посылки и приема сообщений. При этом приходится квантовать алгоритм, обеспечивать своевременную смену информации в буферах, преобразовывать индексы массивов. Все это сильно усложняет программирование и отладку. DSM скрывает от программиста пересылку данных и обеспечивает ему абстракцию разделяемой памяти, к использованию которой он уже привык на мультипроцессорах. Программирование и отладка с использованием DSM гораздо проще.
- 2) В модели передачи сообщений данные перемещаются между двумя различными адресными пространствами. Это делает очень трудным передачу сложных структур данных между процессами. Например, передача данных по ссылке и передача структур данных, содержащих указатели, вызывает серьезные проблемы. В системах с DSM этих проблем нет, что несомненно упрощает разработку распределенных приложений.

# Достоинства DSM

- 3) Объем суммарной физической памяти всех узлов может быть огромным. Эта огромная память становится доступна приложению без издержек, связанных в традиционных системах с дисковыми обменами. Это достоинство становится все весомее в связи с тем, что скорости процессоров и коммуникаций растут быстрее скоростей памяти и дисков.
- 4) DSM-системы могут наращиваться практически беспрепятственно в отличие от систем с разделяемой памятью, т.е. являются масштабируемыми.
- 5) Программы, написанные для мультипроцессоров с общей памятью, могут в принципе без каких-либо изменений выполняться на DSM-системах (по крайней мере, они могут быть легко перенесены на DSM-системы).

# Система с DSM



- Система состоит из однородных базовых модулей (плат), состоящих из небольшого числа процессоров и блока памяти.
- Модули объединены с помощью высокоскоростного коммутатора.
- Поддерживается единое адресное пространство.
- Доступ к локальной памяти в несколько раз быстрее, чем к удаленной.

# Система с DSM



## HPE SGI Altix UV (UltraVioluet) 2000

- 256 Intel® Xeon® processor E5-4600 product family 2.4GHz-3.3GHz - 2048 Cores (4096 Threads)
- 64 ТВ памяти
- NUMALink6 (NL6; 6.7GB/s bidirectional)

# Алгоритмы реализации DSM

При реализации DSM центральными являются следующие вопросы:

- как поддерживать информацию о расположении удаленных данных,
- как снизить при доступе к удаленным данным коммуникационные задержки и большие накладные расходы, связанные с выполнением коммуникационных протоколов,
- как сделать разделяемые данные доступными одновременно на нескольких узлах для того, чтобы повысить производительность системы.

# Алгоритм с централизованным сервером

- Все разделяемые данные поддерживает центральный сервер. Он возвращает данные клиентам по их запросам на чтение, по запросам на запись он корректирует данные и посыпает клиентам в ответ квитанции. Клиенты могут использовать тайм-аут для посылки повторных запросов при отсутствии ответа сервера. Дубликаты запросов на запись могут распознаваться путем нумерации запросов. Если несколько повторных обращений к серверу остались без ответа, приложение получит отрицательный код ответа (это обеспечит клиент).
- Алгоритм прост в реализации, но сервер будет узким местом.
- Можно разделяемые данные распределить между несколькими серверами. В этом случае клиент должен уметь определять, к какому серверу надо обращаться при каждом доступе к разделяемой переменной. Можно, например, распределить между серверами данные в зависимости от их адресов и использовать функцию отображения для определения нужного сервера.
- Независимо от числа серверов, работа с памятью будет требовать коммуникаций и катастрофически замедлится.

# Миграционный алгоритм

- В отличие от предыдущего алгоритма, когда запрос к данным направлялся в место их расположения, в этом алгоритме меняется расположение данных - они перемещаются в то место, где потребовались. Это позволяет последовательные обращения к данным осуществлять локально. Миграционный алгоритм позволяет обращаться к одному элементу данных в любой момент времени только одному узлу.
- Обычно мигрирует целиком страницы или блоки данных, а не запрашиваемые единицы данных. Это позволяет воспользоваться присущей приложениям локальностью доступа к данным для снижения стоимости миграции. Однако, такой подход приводит к трэшингу, когда страницы очень часто мигрируют между узлами при малом количестве обслуживаемых запросов. Причиной этого может быть так называемое "ложное разделение", когда разным процессорам нужны разные данные, но эти данные расположены в одном блоке или странице. Некоторые системы позволяют задать время, в течение которого страница насиливо удерживается в узле для того, чтобы успеть выполнить несколько обращений к ней до ее миграции в другой узел.

# Миграционный алгоритм

- Миграционный алгоритм позволяет интегрировать DSM с виртуальной памятью, обеспечивающейся операционной системой в отдельных узлах. Если размер страницы DSM совпадает с размером страницы виртуальной памяти (или кратен ей), то можно обращаться к разделяемой памяти обычными машинными командами, воспользовавшись аппаратными средствами проверки наличия в оперативной памяти требуемой страницы и замены виртуального адреса на физический. Конечно, для этого виртуальное адресное пространство процессоров должно быть достаточно, чтобы адресовать всю разделяемую память. При этом, несколько процессов в одном узле могут разделять одну и ту же страницу.
- Для определения места расположения блоков данных миграционный алгоритм может использовать сервер, отслеживающий перемещения блоков, либо воспользоваться механизмом подсказок в каждом узле. Возможна и широковещательная рассылка запросов.

# Алгоритм размножения для чтения

- Предыдущий алгоритм позволял обращаться к разделяемым данным в любой момент времени только процессам в одном узле (в котором эти данные находятся). Данный алгоритм расширяет миграционный алгоритм механизмом размножения блоков данных, позволяя либо многим узлам иметь возможность одновременного доступа по чтению, либо одному узлу иметь возможность читать и писать данные (протокол многих читателей и одного писателя).
- При использовании такого алгоритма требуется отслеживать расположение всех блоков данных и их копий. Например, каждый собственник блока может отслеживать расположение его копий.
- Безусловно, производительность повышается за счет возможности одновременного доступа по чтению, но запись требует серьезных затрат для уничтожения всех устаревших копий блока данных или их коррекции. Да и модель многих читателей и одного писателя мало подходит для параллельных программ.

# Алгоритм размножения для чтения и записи

- Этот алгоритм является расширением предыдущего алгоритма. Он позволяет многим узлам иметь одновременный доступ к разделяемым данным на чтение и запись (протокол многих читателей и многих писателей). Поскольку много узлов могут писать данные параллельно, требуется для поддержания согласованности данных контролировать доступ к ним.
- Одним из способов обеспечения согласованности данных является использование специального процесса для упорядочивания модификаций памяти. Все узлы, желающие модифицировать разделяемые данные должны посыпать свои модификации этому процессу. Он будет присваивать каждой модификации очередной номер и рассыпать его широковещательно вместе с модификацией всем узлам, имеющим копию модифицируемого блока данных. Каждый узел будет осуществлять модификации в порядке возрастания их номеров. Разрыв в номерах полученных модификаций будет означать потерю одной или нескольких модификаций. В этом случае узел может запросить недостающие модификации.

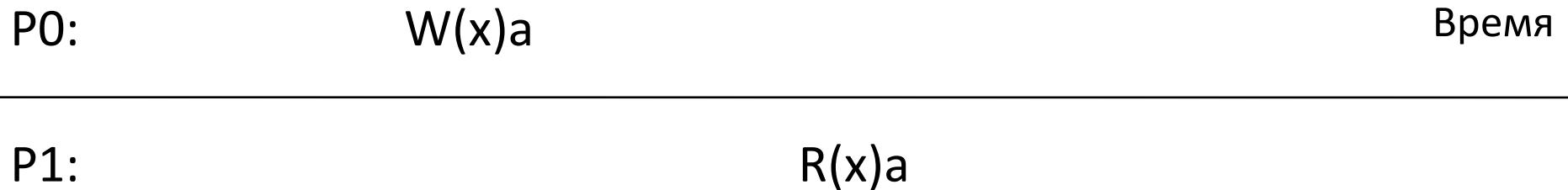
# Модели консистентности

- Модель консистентности представляет собой некоторый договор между программами и памятью, в котором указывается, что при соблюдении программами определенных правил работы с памятью будет обеспечена определенная семантика операций чтения/записи, если же эти правила будут нарушены, то память не гарантирует правильность выполнения операций чтения/записи.
- Далее рассматриваются основные модели консистентности используемые в системах с распределенной памятью.

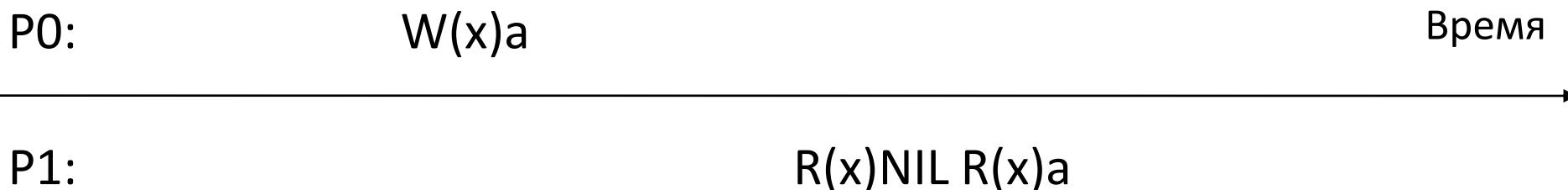
# Строгая консистентность

Операция чтения ячейки памяти с адресом  $X$  должна возвращать значение, записанное самой последней операцией записи с адресом  $X$ , называется моделью **строгой консистентности**.

а) строгая консистентность



б) нестрогая консистентность



# Последовательная консистентность

- Впервые определил Lamport в 1979 г. в контексте совместно используемой памяти для мультипроцессорных систем.
- «Результат выполнения должен быть тот же, как если бы операторы всех процессоров выполнялись бы в некоторой последовательности, в которой операторы каждого индивидуального процессора расположены в порядке, определяемом программой этого процессора»
- Последовательная консистентность не гарантирует, что операция чтения возвратит значение, записанное другим процессом наносекундой или даже минутой раньше, в этой модели только точно гарантируется, что все процессы должны «видеть» одну и ту же последовательность записей в память.

# Последовательная консистентность

а) удовлетворяет последовательной консистентности

P1	<b>W(x)a</b>				
P2		<b>W(x)b</b>			
P3			<b>R(x)b</b>		<b>R(x)a</b>
P4				<b>R(x)b</b>	<b>R(x)a</b>

б) не удовлетворяет последовательной консистентности

P1	<b>W(x)a</b>				
P2		<b>W(x)b</b>			
P3			<b>R(x)b</b>		<b>R(x)a</b>
P4				<b>R(x)a</b>	<b>R(x)b</b>

# Последовательная консистентность

Результат повторного выполнения параллельной программы в системе с последовательной консистентностью может не совпадать с результатом предыдущего выполнения этой же программы, если в программе нет регулирования операций доступа к памяти с помощью механизмов синхронизации.

P1	P2	P3
$x=1;$ <b>Print (y,z);</b>	$y=1;$ <b>Print(x,z);</b>	$z=1;$ <b>Print (x,y);</b>

$x=1;$ <b>Print (y,z);</b>	$x=1;$ <b>y=1;</b>	$y=1;$ <b>z=1;</b>	$y=1;$ <b>x=1;</b>
$y=1;$ <b>Print(x,z);</b>	<b>Print(x,z);</b>	<b>Print (x,y);</b>	$z=1;$ <b>Print(x,z);</b>
<b>Print(x,z);</b>	<b>Print (y,z);</b>	<b>Print(x,z);</b>	<b>Print(x,z);</b>
$z=1;$ <b>Print (y,z);</b>	$z=1;$ <b>Print (x,y);</b>	$x=1;$ <b>Print (y,z);</b>	<b>Print (y,z);</b>
<b>Print (x,y);</b>	<b>Print (x,y);</b>	<b>Print (y,z);</b>	<b>Print (x,y);</b>
<b>001011</b>	<b>101011</b>	<b>0101111</b>	<b>111111</b>

# Последовательная консистентность

Описанный ранее миграционный алгоритм реализует последовательную консистентность.

Последовательная консистентность может быть реализована гораздо более эффективно следующим образом. Страницы, доступные на запись, размножаются, но операции с разделяемой памятью (и чтение, и запись) не должны начинаться на каждом процессоре до тех пор, пока не завершится выполнение предыдущей операции записи, выданной этим процессором, т.е. будут скорректированы все копии соответствующей страницы.

# Последовательная консистентность. Реализация

Централизованный алгоритм. Процесс посылает координатору запрос на модификацию переменной и ждет от него указания о проведении этой модификации. Такое указание координатор рассыпает сразу всем владельцам копий этой переменной. Каждый процесс выполняет эти указания по мере их получения. Поскольку сообщения от координатора приходят каждому процессу в том порядке, в котором они были им посланы, то все процессы корректируют свои копии переменных в этом едином порядке.

Децентрализованный алгоритм. Процесс посылает посредством механизма упорядоченного широковещания (неделимые широковещательные рассылки) указание о модификации переменной всем владельцам копий соответствующей страницы (включая и себя) и ждет получения этого своего собственного указания.

# Причинная консистентность

- Предположим, что процесс P1 модифицировал переменную **x**, затем процесс P2 прочитал **x** и модифицировал **y**. В этом случае модификация **x** и модификация **y** потенциально причинно зависимы, так как новое значение **y** могло зависеть от прочитанного значения переменной **x**. С другой стороны, если два процесса одновременно изменяют значения различных переменных, то между этими событиями нет причинной связи.
- Операции, которые причинно не зависят друг от друга называются параллельными.
- Причинная модель консистентности памяти определяется следующим условием: Последовательность операций записи, которые потенциально причинно зависимы, должна наблюдаться всеми процессами системы одинаково, параллельные операции записи могут наблюдаться разными процессами в разном порядке.

# Причинная консистентность

P1	W(x)a				
P2		R(x)a	W(x)b		
P3				R(x)b	R(x)a
P4				R(x)a	R(x)b

Нарушение модели причинной консистентности

Корректная последовательность для модели причинной консистентности

P1	W(x)a			W(x)c		
P2		R(x)a	W(x)b			
P3		R(x)a			R(x)c	R(x)b
P4		R(x)a			R(x)b	R(x)c

Определение потенциальной причинной зависимости может осуществляться компилятором посредством анализа зависимости операторов программы по данным.

# Причинная консистентность. Реализация

При реализации причинной консистентности в случае размножения страниц выполнение записи в общую память требует ожидания выполнения только тех предыдущих операций записи, от которых эта запись потенциально причинно зависит. Параллельные операции записи не задерживают выполнение друг друга (и не требуют неделимости широковещательных рассылок всем владельцам копий страницы).

Реализация причинной консистентности может осуществляться следующим образом:

- все модификации переменных на каждом процессоре нумеруются;
- всем процессорам вместе со значением модифицируемой переменной рассыдается номер этой модификации на данном процессоре, а также номера модификаций всех процессоров, известных данному процессору к этому моменту;
- выполнение любой модификации на каждом процессоре задерживается до тех пор, пока он не получит и не выполнит все те модификации других процессоров, о которых было известно процессору - автору задерживаемой модификации.

# PRAM-консистентность

Операции записи, выполняемые одним процессором, видны всем остальным процессорам в том порядке, в каком они выполнялись, но операции записи, выполняемые разными процессорами, могут быть видны в произвольном порядке.

Записи выполняемые одним процессором могут быть конвейеризованы: выполнение операций с общей памятью можно начинать не дожидаясь завершения предыдущих операций записи в память.

P1:     $W(x)1$

---

P2:                   $R(x)1$      $W(x)2$

---

P3:                   $R(x)1$      $R(x)2$

---

P4:                   $R(x)2$      $R(x)1$

Допустимая последовательность для PRAM-консистентности

# PRAM-консистентность

Преимущество модели PRAM консистентности заключается в простоте ее реализации, поскольку операции записи на одном процессоре могут быть конвейеризованы: можно продолжать выполнение процесса и выполнять другие операции с общей памятью не дожидаясь завершения предыдущих операций записи (модификации всех копий страниц, например), необходимо только быть уверенным, что все процессоры увидят эти записи в одном и том же порядке.

PRAM консистентность может приводить к результатам, противоречащим интуитивному представлению.

Пример:

Процесс P1

.....

$a = 1;$

if ( $b == 0$ ) kill (P2);

.....

Оба процесса могут быть убиты, что невозможно при последовательной консистентности.

Процесс P2

.....

$b = 1;$

if ( $a == 0$ ) kill (P1);

.....

# Процессорная консистентность

PRAM-консистентность + когерентность памяти

Для каждой переменной  $x$  есть общее согласие относительно порядка, в котором процессоры модифицируют эту переменную, операции записи в разные переменные - параллельны.

Таким образом, к упорядочиванию записей каждого процессора добавляется упорядочивание записей в переменные или группы переменных (например, находящихся в независимых блоках памяти).

## Реализация.

За каждую группу переменных отвечает свой координатор, который получает от процессов запросы на модификацию и рассыпает всем указания о проведении модификации. Чтобы не нарушить порядок получения процессами указаний о модификациях различных переменных, запрошенных одним процессом у разных координаторов, надо каждому процессу нумеровать свои модификации, и эти номера должны рассыпаться всем вместе с указаниями о проведении модификаций. Тогда любой процесс, получающий указание о проведении модификации, может задержать его выполнение до получения недостающих указаний о предшествующих модификациях соответствующего процесса.

# Слабая консистентность

- Пусть процесс в критической секции циклически читает и записывает значение некоторых переменных. Даже, если остальные процессоры и не пытаются обращаться к этим переменным до выхода первого процесса из критической секции, для удовлетворения требований рассматриваемых ранее моделей консистентности они должны видеть все записи первого процессора в порядке их выполнения, что, естественно, совершенно не нужно.
- Наилучшее решение в такой ситуации - это позволить первому процессу завершить выполнение критической секции и, только после этого, переслать остальным процессам значения модифицированных переменных, не заботясь о пересылке промежуточных результатов.

# Слабая консистентность

Модель слабой консистентности, основана на выделении среди переменных специальных синхронизационных переменных и описывается следующими правилами:

1. Доступ к синхронизационным переменным определяется моделью последовательной консистентности;
2. Доступ к синхронизационным переменным запрещен (задерживается), пока не выполнены все предыдущие операции записи;
3. Доступ к данным (запись, чтение) запрещен, пока не выполнены все предыдущие обращения к синхронизационным переменным.

# Слабая консистентность

- Первое правило определяет, что все процессы видят обращения к синхронизационным переменным в определенном (одном и том же) порядке.
- Второе правило гарантирует, что выполнение процессором операции обращения к синхронизационной переменной возможно только после выталкивания конвейера (полного завершения выполнения на всех процессорах всех предыдущих операций записи переменных, выданных данным процессором).
- Третье правило определяет, что при обращении к обычным (не синхронизационным) переменным на чтение или запись, все предыдущие обращения к синхронизационным переменным должны быть выполнены полностью. Выполнив синхронизацию перед обращением к общей переменной, процесс может быть уверен, что получит правильное значение этой переменной.

# Слабая консистентность

P1	W(x)a	W(x)b	S			
P2				R(x)a	R(x)b	S
P3				R(x)b	R(x)a	S

Допустимая последовательность событий

Недопустимая последовательность событий

P1	W(x)a	W(x)b	S		
P2				S	R(x)a

# Консистентность по выходу

В системе со слабой консистентностью возникает проблема при обращении к синхронизационной переменной: система не имеет информации о цели этого обращения - или процесс завершил модификацию общей переменной, или готовится прочитать значение общей переменной. Для более эффективной реализации модели консистентности система должна различать две ситуации: вход в критическую секцию и выход из нее. В модели консистентности по выходу введены специальные функции обращения к синхронизационным переменным:

- 1) ACQUIRE - захват синхронизационной переменной, информирует систему о входе в критическую секцию;
- 2) RELEASE - освобождение синхронизационной переменной, определяет завершение критической секции.

Захват и освобождение используется для организации доступа не ко всем общим переменным, а только к тем, которые защищаются данной синхронизационной переменной. Такие общие переменные называют защищенными переменными.



# Консистентность по выходу

- Существует модификация консистентности по выходу - «ленивая». В отличие от описанной («энергичной») консистентности по выходу, она не требует выталкивания всех модифицированных данных при выходе из критической секции. Вместо этого, при запросе входа в критическую секцию процессу передаются текущие значения защищенных разделяемых переменных (например, от процесса, который последним находился в критической секции, охраняемой этой синхронизационной переменной).
- При повторных входах в критическую секцию того же самого процесса не требуется никаких обменов сообщениями.
- Для того, чтобы узнать, какие переменные защищаются конкретной синхронизационной переменной, нужно фиксировать все переменные, изменяемые внутри соответствующих критических секций.

# Консистентность по входу

Эта консистентность представляет собой еще один пример модели консистентности, которая ориентирована на использование критических секций. Так же, как и в предыдущей модели, эта модель консистентности требует от программистов (или компиляторов) использование механизма захвата/освобождения для выполнения критических секций.

Однако в этой модели требуется, чтобы каждая общая переменная была явна связана с некоторой синхронизационной переменной (или с несколькими синхронизационными переменными), при этом, если доступ к элементам массива, или различным отдельным переменным, может производиться независимо (параллельно), то эти элементы массива (общие переменные) должны быть связаны с разными синхронизационными переменными. Таким образом, вводится явная связь между синхронизационными переменными и общими переменными, которые они охраняют.

# Консистентность по входу

Кроме того, критические секции, охраняемые одной синхронизационной переменной, могут быть двух типов:

- секция с монопольным доступом (для модификации переменных);
- секция с немонопольным доступом (для чтения переменных).

Каждая синхронизационная переменная имеет временного владельца - последний процесс, захвативший доступ к этой переменной. Этот владелец может в цикле выполнять критическую секцию, не посыпая при этом сообщений другим процессорам.

Процесс, который в данный момент не является владельцем синхронизационной переменной, но требующий ее захвата, должен послать запрос текущему владельцу этой переменной для получения права собственности на синхронизационную переменную и значений охраняемых ею общих переменных.

Разрешена ситуация, когда синхронизационная переменная имеет несколько владельцев, но только в том случае, если связанные с этой переменной общие данные используются только для чтения.

## Консистентность по входу

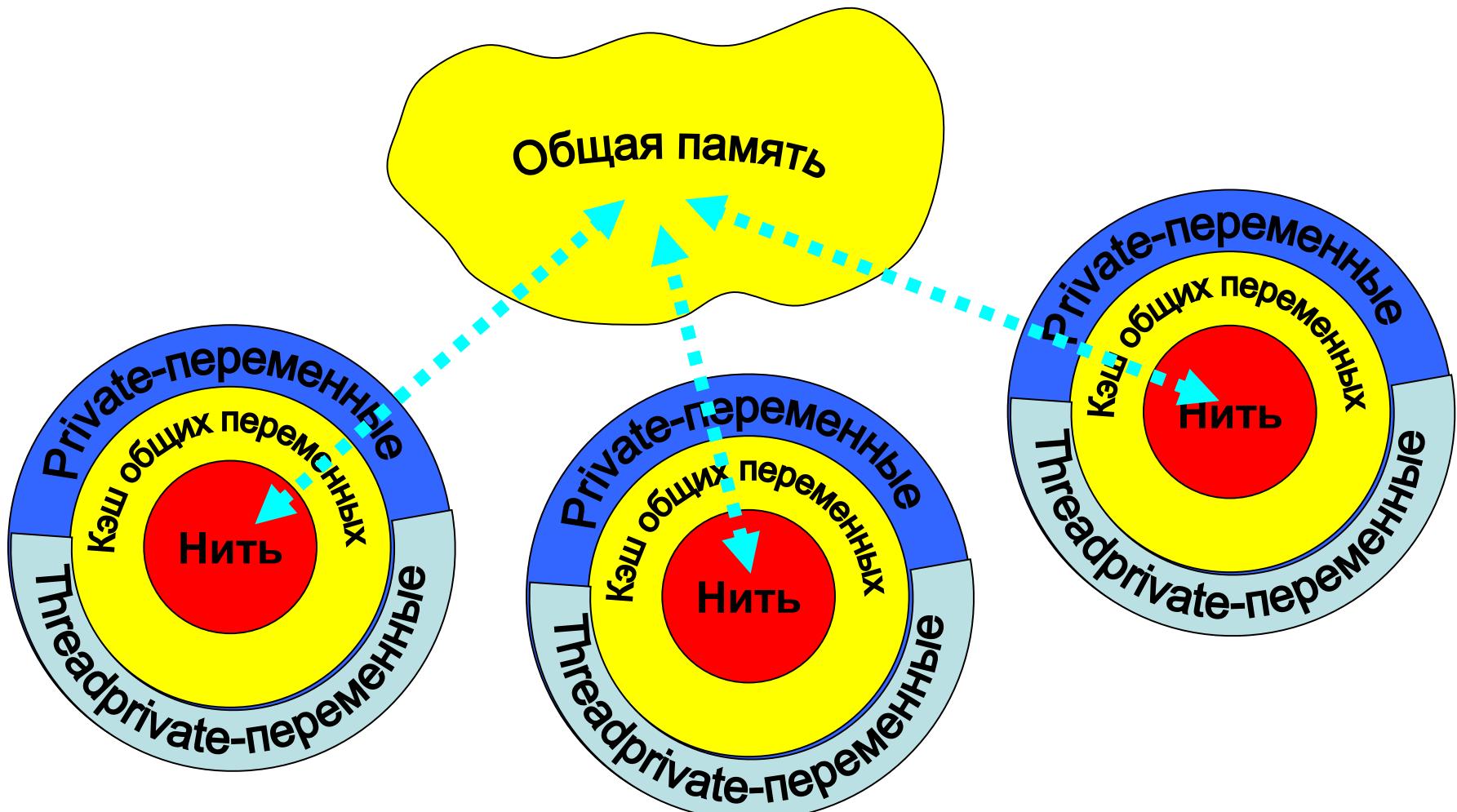
- Процесс не может захватить синхронизационную переменную до того, пока не обновлены все переменные этого процесса, охраняемые захватываемой синхронизационной переменной;
- Процесс не может захватить синхронизационную переменную в монопольном режиме (для модификации охраняемых данных), пока другой процесс, владеющий этой переменной (даже в немонопольном режиме), не освободит ее;
- Если какой-то процесс захватил синхронизационную переменную в монопольном режиме, то ни один процесс не сможет ее захватить даже в немонопольном режиме до тех пор, пока первый процесс не освободит эту переменную, и будут обновлены текущие значения охраняемых переменных в процессе, запрашивающем синхронизационную переменную.

# Сравнение моделей консистентности

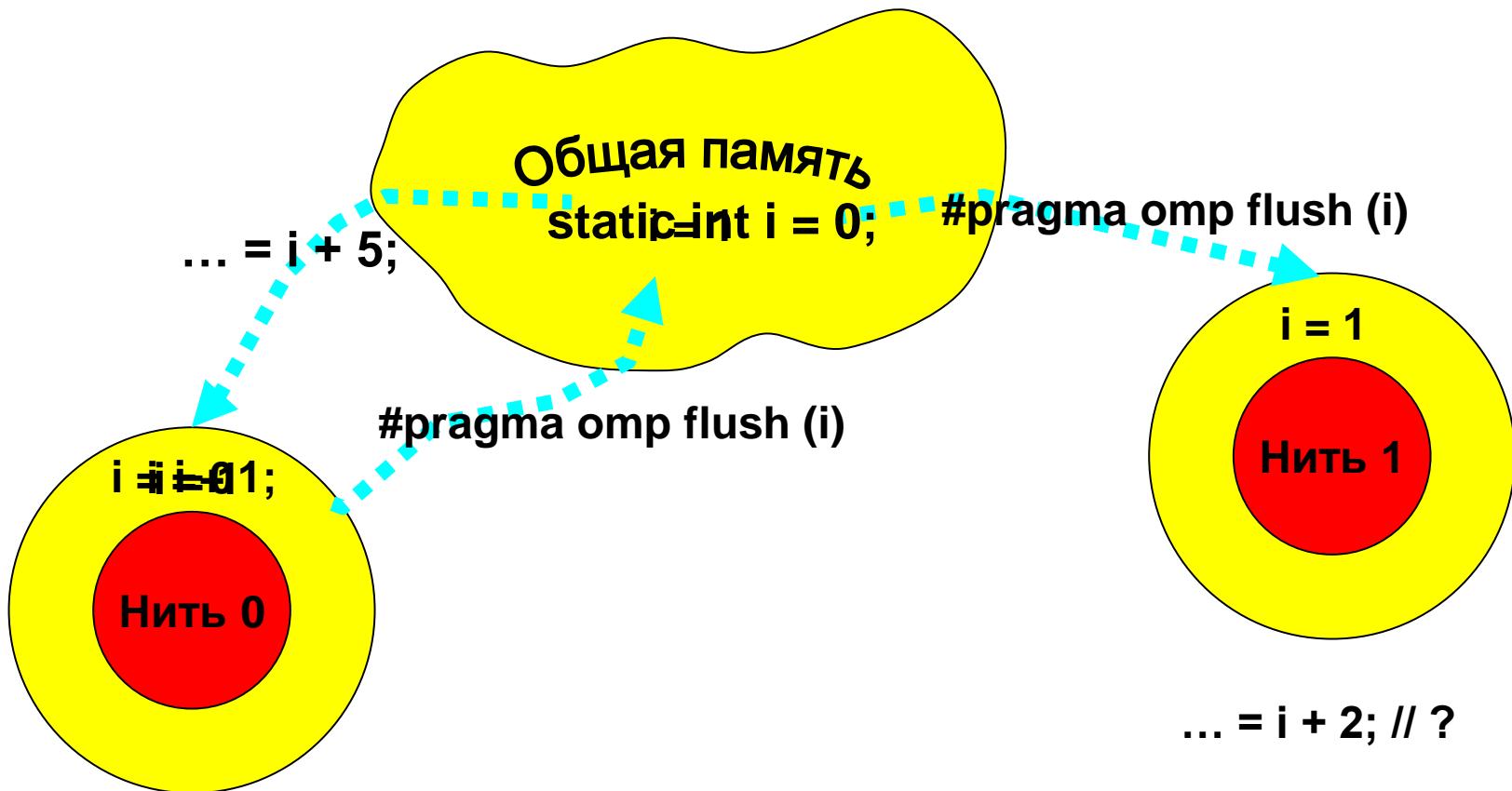
Консистентность	Описание
Строгая	Упорядочение всех доступов к разделяемым данным по абсолютному времени
Последовательная	Все процессы видят все записи разделяемых данных в одном и том же порядке
Причинная	Все процессы видят все причинно-связанные записи данных в одном и том же порядке
Процессорная	PRAM-консистентность + когерентность памяти
PRAM	Все процессоры видят записи любого процессора в одном и том же порядке

Консистентность	Описание
Слабая	Разделяемые данные можно считать консистентными только после выполнения синхронизации
По выходу	Разделяемые данные, изменяемые в критической секции, становятся консистентными после выхода из нее.
По входу	Разделяемые данные, связанные с монопольной или немонопольной критической секцией, становятся консистентными при входе в нее

# Консистентность памяти в OpenMP



# Консистентность памяти в OpenMP



# Консистентность памяти в OpenMP

Корректная последовательность работы нитей с переменной:

- Нить0 записывает значение переменной - write(var)
- Нить0 выполняет операцию синхронизации – flush (var)
- Нить1 выполняет операцию синхронизации – flush (var)
- Нить1 читает значение переменной – read (var)

Директива flush:

**#pragma omp flush [(list)]** - для Си

**!\$omp flush [(list)]** - для Фортран

# Консистентность памяти в OpenMP

`#pragma omp flush [(список переменных)]`

По умолчанию все переменные приводятся в консистентное состояние (`#pragma omp flush`):

- при барьерной синхронизации;
- при входе и выходе из конструкций `parallel`, `critical` и `ordered`;
- при выходе из конструкций распределения работ (`for`, `single`, `sections`, `workshare`), если не указана клауза `nowait`;
- при вызове `omp_set_lock` и `omp_unset_lock`;
- при вызове `omp_test_lock`, `omp_set_nest_lock`, `omp_unset_nest_lock` и `omp_test_nest_lock`, если изменилось состояние семафора.

При входе и выходе из конструкции `atomic` выполняется `#pragma omp flush(x)`, где `x` – переменная, изменяемая в конструкции `atomic`.

# Консистентность памяти в OpenMP

1. Если пересечение множеств переменных, указанных в операциях flush, выполняемых различными нитями не пустое, то результат выполнения операций flush будет таким, как если бы эти операции выполнялись в некоторой последовательности (единой для всех нитей).
2. Если пересечение множеств переменных, указанных в операциях flush, выполняемых одной нитью не пустое, то результат выполнения операций flush, будет таким, как если бы эти операции выполнялись в порядке определяемом программой.
3. Если пересечение множеств переменных, указанных в операциях flush, пустое, то операции flush могут выполняться независимо (в любом порядке).

# Технология Intel Cluster OpenMP

В 2006 году в Intel® компиляторах версии 9.1 появилась поддержка Cluster OpenMP.

Технология Cluster OpenMP поддерживает выполнение OpenMP программ на нескольких вычислительных узлах, объединенных сетью.

Базируется на программной реализации DSM (Thread Marks software by Rice University).

Для компилятора Intel® C++:

- `icc -cluster-openmp options source-file`
- `icc -cluster-openmp-profile options source-file`

Для компилятора Intel® Fortran:

- `ifort -cluster-openmp options source-file`
- `ifort -cluster-openmp-profile options source-file`

`kmp_cluster.ini`

`--hostlist=home,remote --process_threads=2`

# Технология Intel Cluster OpenMP

**#pragma intel omp sharable ( variable [, variable ...] )** – для Си и Си++

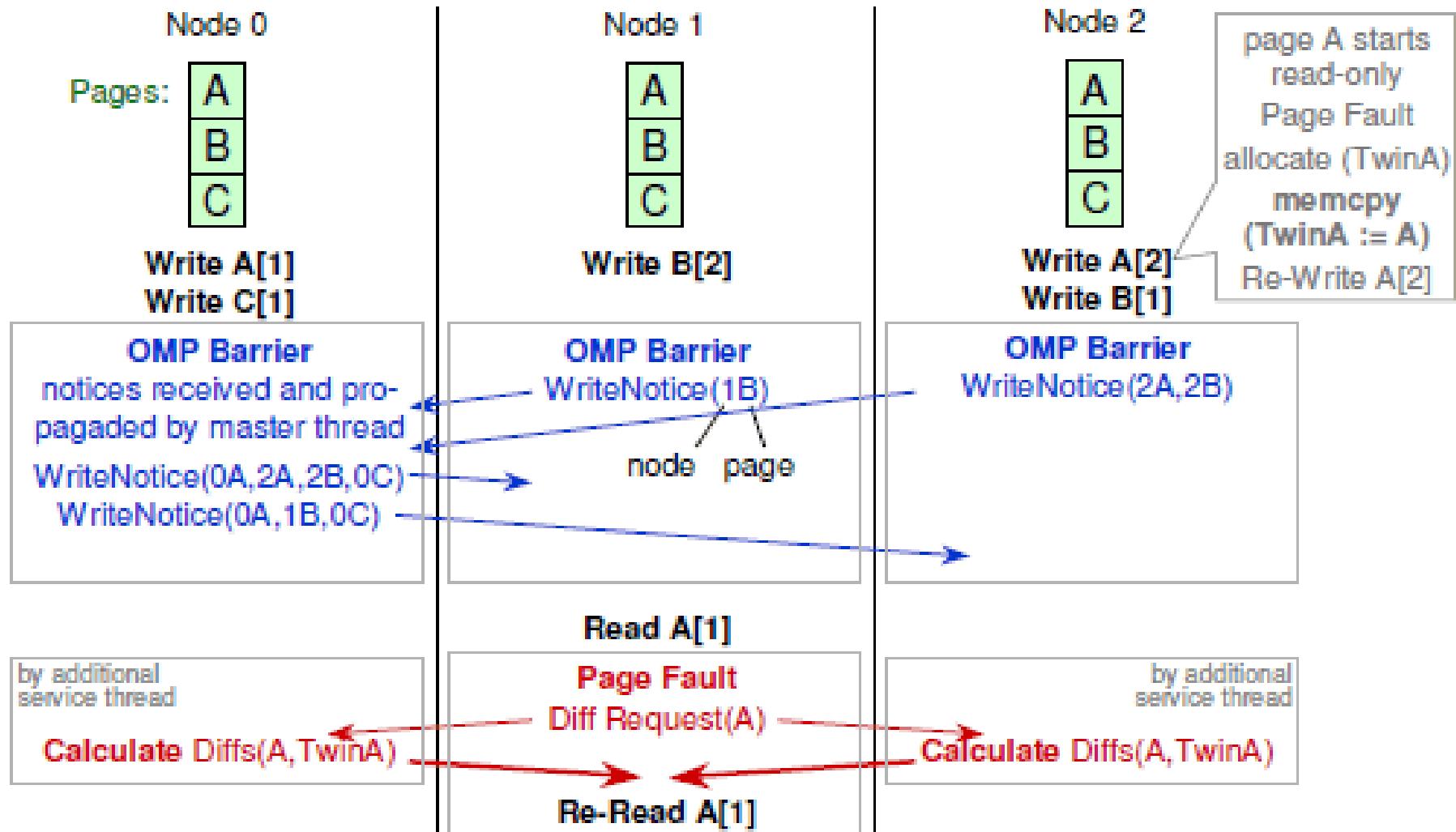
**!dir\$ omp sharable ( variable [, variable ...] )** - для Фортрана

определяют переменные, которые должны быть помещены в  
**Distributed Virtual Shared Memory**

В компиляторе существуют опции, которые позволяют изменить класс  
переменных, не изменяя текст программы:

- [-no]-clomp-sharable-argexprs
- [-no]-clomp-sharable-commons
- [-no]-clomp-sharable-localsaves
- [-no]-clomp-sharable-modvars

# Технология Intel Cluster OpenMP



# Критическая секция

a = b = tmp = 0

**thread 1**

b = 1

flush(b)

flush(a)

tmp = a

if (tmp == 0) then

*protected section*

end if

**thread 2**

a = 1

flush(a)

flush(b)

tmp = b

if (tmp == 0) then

*protected section*

end if

# Критическая секция

a = b = tmp = 0

**thread 1**

b = 1

flush(a,b)

tmp = a

if (tmp == 0) then

*protected section*

end if

**thread 2**

a = 1

flush(a,b)

tmp = b

if (tmp == 0) then

*protected section*

end if

# OpenMP версии 5.0

**#pragma omp flush [memory-order-clause] [(список переменных)]**

где *memory-order-clause*:

**acq\_rel**

**release**

**acquire**

# Критическая секция

```
#include <stdio.h>
#include <omp.h>
int main() {
    int x = 0, y = 0;
    #pragma omp parallel num_threads(2)
    {
        if (omp_get_thread_num() == 0) {
            x = 10;
            #pragma omp atomic write release
            y = 1;
        } else {
            int tmp = 0;
            while (tmp == 0) {
                #pragma omp atomic read acquire
                tmp = y;
            }
            printf("x = %d\n", x);
        }
    }
    return 0;
}
```

# Надежность в распределенных системах

# Отказы в распределенных системах

- Отказом системы называется поведение системы, не удовлетворяющее ее спецификациям. Последствия отказа могут быть различными.
- Отказ системы может быть вызван отказом (неверным срабатыванием) каких-то ее компонентов (процессор, память, устройства ввода/вывода, линии связи, или программное обеспечение).
- Отказ компонента может быть вызван ошибками при конструировании, при производстве или программировании. Он может быть также вызван физическим повреждением, изнашиванием оборудования, некорректными входными данными, ошибками оператора, и многими другими причинами.

# Отказы в распределенных системах

- Отказы могут быть случайными, периодическими или постоянными.
- Случайные отказы (сбои) при повторении операции исчезают.
- Причиной такого сбоя может служить, например, электромагнитная помеха от проезжающего мимо трамвая. Другой пример - редкая ситуация в последовательности обращений к операционной системе от разных задач.
- Периодические отказы повторяются часто в течение какого-то времени, а затем могут долго не происходить. Примеры - плохой контакт, некорректная работа ОС после обработки аварийного завершения задачи.
- Постоянные (устойчивые) отказы не прекращаются до устранения их причины - разрушения диска, выхода из строя микросхемы или ошибки в программе.

# Отказы в распределенных системах

- Отказы по характеру своего проявления подразделяются на «византийские» (система активна и может проявлять себя по-разному, даже злонамеренно) и «пропажа признаков жизни» (частичная или полная). Первые распознать гораздо сложнее, чем вторые. Свое название они получили по имени Византийской империи (330-1453 гг.), где расцветали конспирация, интриги и обман.
- Для обеспечения надежного решения задач в условиях отказов системы применяются два принципиально различающихся подхода - восстановление решения после отказа системы (или ее компонента) и предотвращение отказа системы (отказоустойчивость).

# Восстановление после отказа

- Восстановление может быть прямым (без возврата к прошлому состоянию) и возвратное.
- Прямое восстановление основано на своевременном обнаружении сбоя и ликвидации его последствий путем приведения некорректного состояния системы в корректное. Такое восстановление возможно только для определенного набора заранее предусмотренных сбоев.
- При возвратном восстановлении происходит возврат процесса (или системы) из некорректного состояния в некоторое из предшествующих корректных состояний.

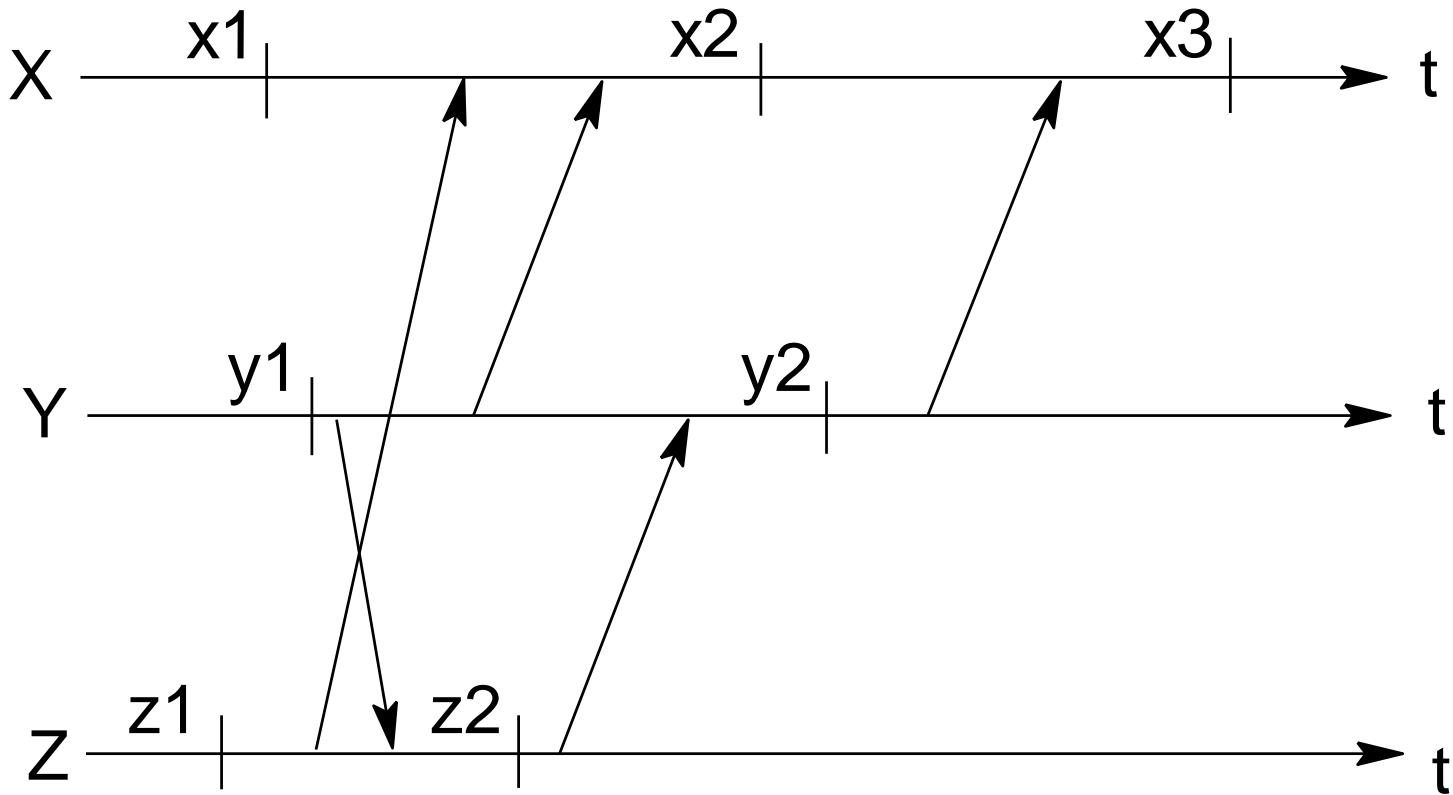
# Возвратное восстановление

При возвратном восстановлении возникают следующие проблемы.

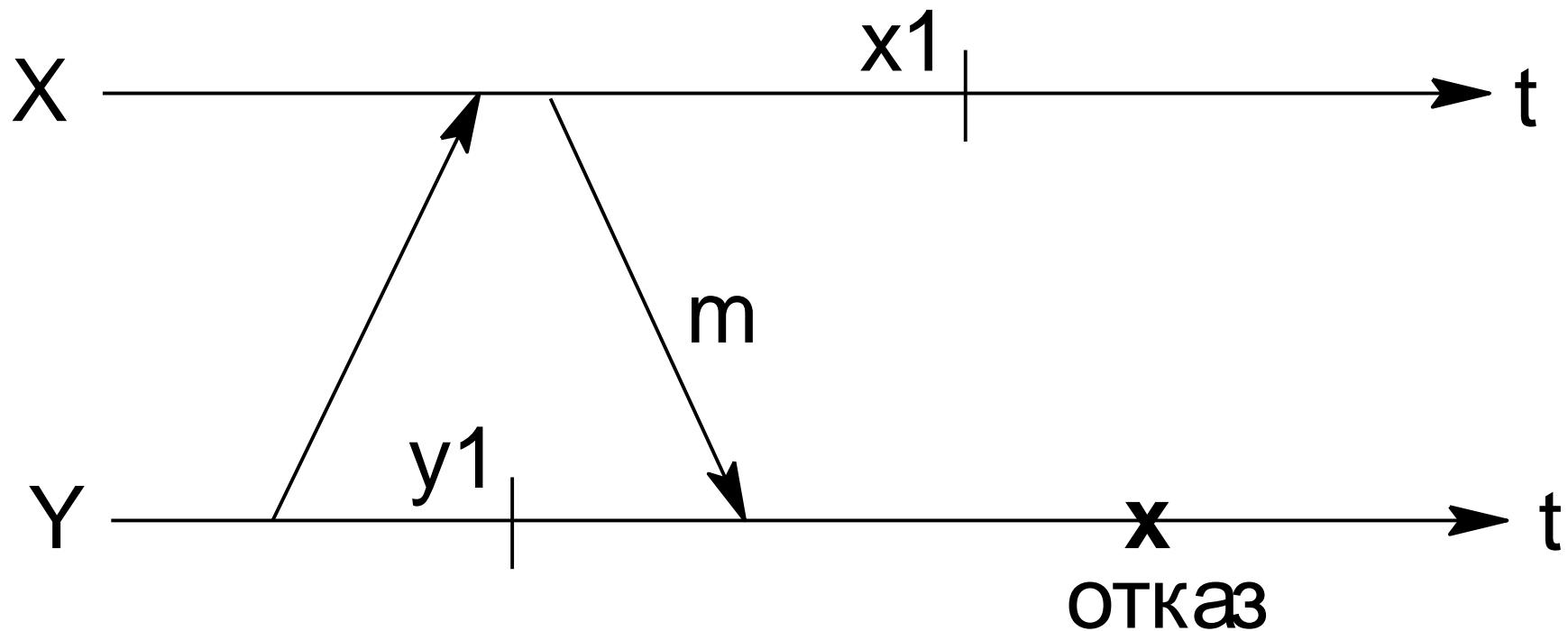
- Потери производительности, вызванные запоминанием состояний, восстановлением запомненного состояния и повторением ранее выполненной работы, могут быть слишком высоки.
- Нет гарантии, что сбой снова не повторится после восстановления.
- Для некоторых компонентов системы восстановление в предшествующее состояние может быть невозможно (торговый автомат).

Для восстановления состояния в традиционных ЭВМ применяются два метода (и их комбинация), основанные на промежуточной фиксации состояния либо ведении журнала выполняемых операций. Они различаются объемом запоминаемой информацией и временем, требуемым для восстановления.

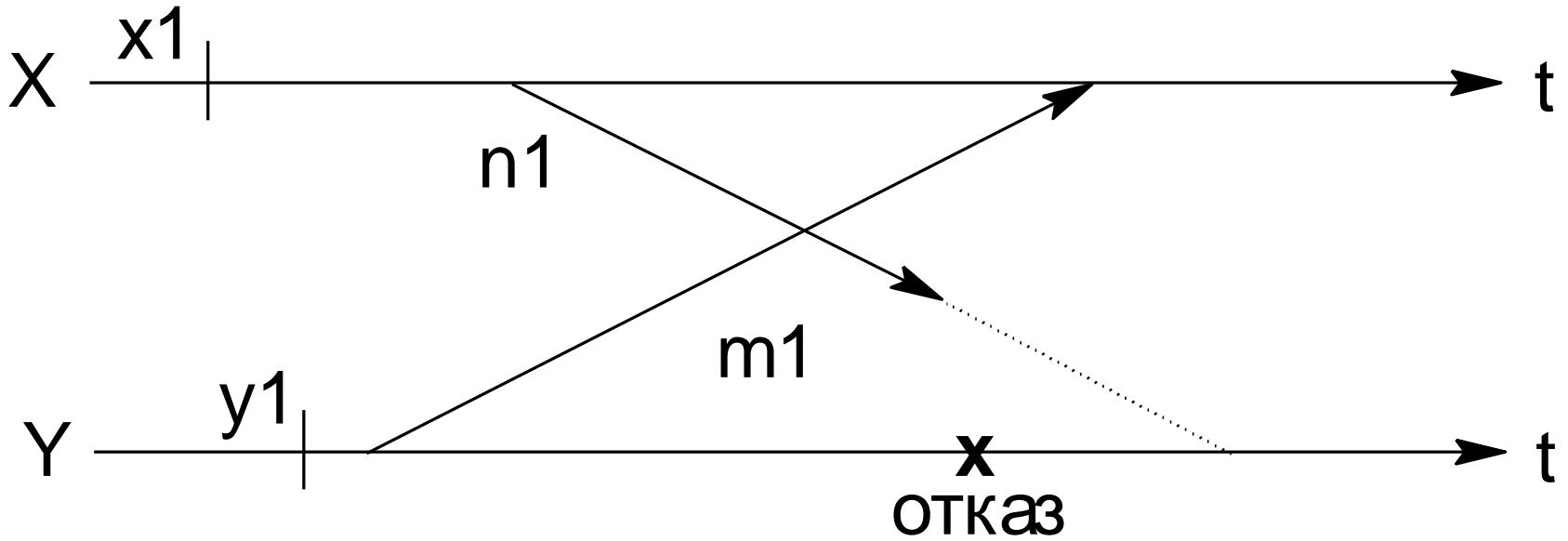
# Сообщения сироты и эффект домино



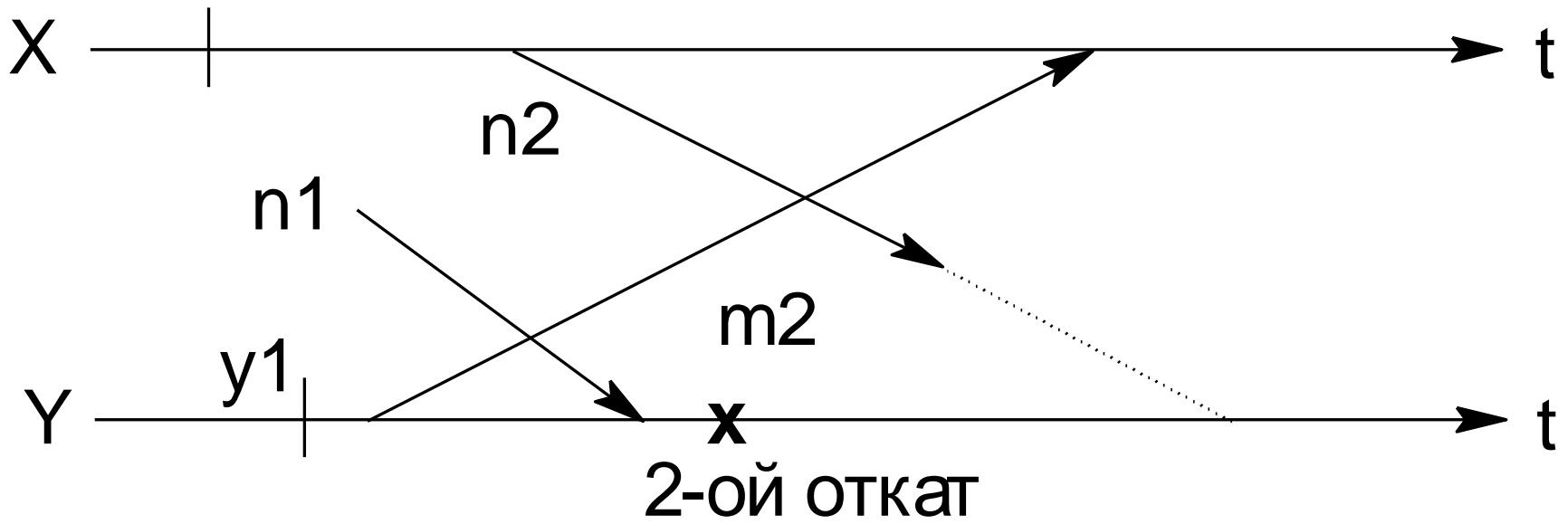
# Потеря сообщений



# Проблема бесконечного восстановления



# Проблема бесконечного восстановления



$n_1$  – сообщение-призрак

# Консистентное множество контрольных точек

- Показанные ранее трудности показывают, что глобальная контрольная точка, состоящая из произвольной совокупности локальных контрольных точек, не обеспечивает восстановления взаимодействующих процессов.
- Для распределенных систем запоминание согласованного глобального состояния является серьезной теоретической проблемой.
- Множество контрольных точек называется **строго консистентным**, если во время его фиксации никаких обменов между процессами не было. Оно соответствует понятию строго консистентного глобального состояния, когда все посланные сообщения получены и нет никаких сообщений в каналах связи.
- Множество контрольных точек называется **консистентным**, если для любой зафиксированной операции приема сообщения, соответствующая операция посылки также зафиксирована (нет сообщений-сирот).

# Простой метод фиксации консистентного множества контрольных точек

- Фиксация локальной контрольной точки после каждой операции посылки сообщения. При этом посылка сообщения и фиксация должны быть единой неделимой операцией (транзакцией). Множество последних локальных контрольных точек является консистентным (но не строго консистентным).
- Чтобы избежать потерь сообщений при восстановлении с использованием консистентного множества контрольных точек необходимо повторить отправку тех сообщений, квитанции о получении которых стали недействительными в результате отката. Используя временные метки сообщений можно распознавать сообщения-призраки и избежать бесконечного восстановления.

# Синхронная фиксация контрольных точек и восстановление

К распределенной системе алгоритм предъявляет следующие требования.

- 1) Процессы взаимодействуют посредством посылки сообщений через коммуникационные каналы.
- 2) Каналы работают по алгоритму FIFO. Коммуникационные протоколы точка-точка гарантируют невозможность пропажи сообщений из-за ошибок коммуникаций или отката к контрольной точке.

Другой способ обеспечения этого - использование стабильной памяти для журнала посылаемых сообщений и фиксации идентификатора последнего полученного по каналу сообщения.

# **Синхронная фиксация контрольных точек и восстановление**

## **1-ая фаза.**

- Инициатор фиксации (процесс  $P_i$ ) создает пробную контрольную точку и просит все остальные процессы сделать то же самое. При этом процессу запрещается посылать неслужебные сообщения после того, как он сделает пробную контрольную точку. Каждый процесс извещает  $P_i$  о том, сделал ли он пробную контрольную точку. Если все процессы сделали пробные контрольные точки, то  $P_i$  принимает решение о превращении пробных точек в постоянные. Если какой-либо процесс не смог сделать пробную точку, то принимается решение об отмене всех пробных точек.

# Синхронная фиксация контрольных точек и восстановление

## 2-ая фаза.

- Рі информирует все процессы о своем решении. В результате либо все процессы будут иметь новые постоянные контрольные точки, либо ни один из процессов не создаст новой постоянной контрольной точки. Только после выполнения принятого процессом Рі решения все процессы могут посылать сообщения.
- Корректность алгоритма очевидна, поскольку созданное всеми множество постоянных контрольных точек не может содержать не зафиксированных операций посылки сообщений.
- Оптимизация: если процесс не посыпал сообщения с момента фиксации предыдущей постоянной контрольной точки, то он может не создавать новую.

# Синхронная фиксация контрольных точек и восстановление

Алгоритм восстановления предполагает, что его инициирует один процесс и он не будет выполняться параллельно с алгоритмом фиксации.

Выполняется в две фазы.

- 1) Инициатор отката спрашивает остальных, готовы ли они откатываться. Когда все будут готовы к откату, то он принимает решение об откате.
- 2) Pi сообщает всем о принятом решении. Получив это сообщение, каждый процесс поступает указанным образом. С момента ответа на опрос готовности и до получения принятого решения процессы не должны посыпать сообщения (нельзя же посыпать сообщение процессу, который уже мог успеть откатиться).

# Асинхронная фиксация контрольных точек и восстановление

Синхронная фиксация упрощает восстановление, но связана с большими накладными расходами:

- 1) Дополнительные служебные сообщения для реализации алгоритма.
- 2) Синхронизационная задержка - нельзя посыпать неслужебные сообщения во время работы алгоритма.

Если отказы редки, то указанные потери совсем не оправданы.

Фиксация может производиться асинхронно.

В этом случае множество контрольных точек может быть неконсистентным.

# Асинхронная фиксация контрольных точек и восстановление

При откате происходит поиск подходящего консистентного множества путем поочередного отката каждого процесса в ту точку, в которой зафиксированы все посланные им и полученные другими сообщения (для ликвидации сообщений-сирот).

Алгоритм опирается на наличие в стабильной памяти для каждого процесса журнала, отслеживающего номера посланных и полученных им сообщений, а также на некоторые предположения об организации взаимодействия процессов, необходимые для исключения «эффекта домино» (например, организация приложения по схеме сообщение-реакция-ответ).

# Отказоустойчивость

- Изложенные выше методы восстановления после отказов для некоторых систем непригодны (управляющие системы, транзакции в on-line режиме) из-за прерывания нормального функционирования.
- Чтобы избежать этих неприятностей, создают системы, устойчивые к отказам. Такие системы либо маскируют отказы, либо ведут себя в случае отказа заранее определенным образом (пример - изменения, вносимые транзакцией в базу данных, становятся невидимыми при отказе).
- Два механизма широко используются при обеспечении отказоустойчивости - протоколы голосования и протоколы принятия коллективного решения.

# Протоколы принятия коллективного решения

Разделяются на два класса.

*Протоколы принятия единого решения*, в которых все исполнители являются исправными и должны либо все принять, либо все не принять заранее предусмотренное решение.

Примерами такого решения являются решение о завершении итерационного цикла при достижении всеми необходимой точности, решение о реакции на отказ (этот протокол уже знаком нам - он использовался для принятия решения об откате всех процессов к контрольным точкам).

*Протоколы принятия согласованных решений* на основе полученных друг от друга данных. При этом необходимо всем исправным исполнителям получить достоверные данные от остальных исправных исполнителей, а данные от неисправных исполнителей проигнорировать.

# Отказоустойчивость

Ключевой подход для обеспечения отказоустойчивости - избыточность (оборудования, процессов, данных).

- **Использование режима «горячего резерва»** (второй пилот, резервное ПО).

Проблема переключения на резервный исполнитель.

- **Использование активного размножения.**

Наглядный пример - тройное дублирование аппаратуры в бортовых компьютерах и голосование при принятии решения.

Другие примеры – размножение страниц в DSM и размножение файлов в распределенных файловых системах.

# Алгоритмы голосования

- Общая схема использования голосования при размножении файлов может быть представлена следующим образом.
- Файл может модифицироваться разными процессами только последовательно (при открытии файла на запись процесс-писатель будет ждать закрытия файла другим писателем или всеми читателями), а читаться всеми одновременно (протокол писателей-читателей). Все модификации файла нумеруются и каждая копия файла характеризуется номером версии – количеством ее модификаций.
- Каждой копии приписано некоторое количество голосов  $V_i$ .
- Пусть общее количество приписанных всем копиям голосов равно  $V$
- Определяется кворум записи  $V_w$  и кворум чтения  $V_r$  так, что  $V_w + V_r > V$

# Алгоритмы голосования

- Для записи информации в файл писатель рассыпает ее всем владельцам копий файла и должен получить  $V_w$  голосов от тех, кто успешно выполнил запись.
- Для получения права на чтение читателю достаточно получить необходимое число голосов ( $V_r$ ) от любых серверов.
- Кворум чтения выбран так, что хотя бы один из тех серверов, от которых получено разрешение, является владельцем текущей копии файла. За чтением информации из файла читатель может обратиться к любому владельцу текущей копии файла.
- Описанная схема базируется на **статическом распределении голосов**. Различие в голосах, приписанных разным серверам, позволяет учесть их особенности (надежность, эффективность). Еще большую гибкость предоставляет метод **динамического перераспределения голосов**.
- Для того, чтобы выход из строя некоторых серверов не привел к ситуации, когда невозможно получить кворум, применяется механизм **изменения состава голосующих**.

# Протоколы принятия единого решения

Рассмотрим известную «проблему двух армий».

- Армия зеленых численностью 5000 воинов располагается в долине.
- Две армии синих численностью по 3000 воинов находятся далеко друг от друга в горах, окружающих долину. Если две армии синих одновременно атакуют зеленых, то они победят. Если же в сражение вступит только одна армия синих, то она будет полностью разбита.
- Предположим, что командир 1-ой синей армии генерал Александр посыпает (с посыльным) сообщение командиру 2-ой синей армии генералу Михаилу «Я имею план - давай атаковать завтра на рассвете». Посыльный возвращается к Александру с ответом Михаила - «Отличная идея, Саша. Увидимся завтра на рассвете». Александр приказывает воинам готовиться к атаке на рассвете.

# Протоколы принятия единого решения

- Однако, чуть позже Александр вдруг осознает, что Михаил не знает о возвращении посыльного и поэтому может не отважиться на атаку. Тогда он отправляет посыльного к Михаилу чтобы подтвердить, что его (Михаила) сообщение получено Александром и атака должна состояться.
- Посыльный прибыл к Михаилу, но теперь тот боится, что не зная о прибытии посыльного Александр может не решиться на атаку. И т.д. Ясно, что генералы никогда не достигнут согласия.
- Предположим, что такой протокол согласия с конечным числом сообщений существует. Удалив избыточные последние сообщения, получим минимальный протокол. Самое последнее сообщение является существенным (поскольку протокол минимальный). Если это сообщение не дойдет по назначению, то войны не будет. Но тот, кто послал это сообщение, не знает, дошло ли оно. Следовательно, он не может считать протокол завершенным и не может принять решение об атаке. Даже с надежными процессорами (генералами), принятие единого решения невозможно при ненадежных коммуникациях.

# Задача «Византийских генералов»

- В этой задаче армия зеленых находится в долине, а  $n$  синих генералов возглавляют свои армии, расположенные в горах. Связь осуществляется по телефону и является надежной, но из  $n$  генералов  $m$  являются предателями. Предатели активно пытаются воспрепятствовать согласию лояльных генералов.
- Согласие в данном случае заключается в следующем. Каждый генерал знает, сколько воинов находится под его командой. Ставится цель, чтобы все лояльные генералы узнали численности всех лояльных армий, т.е. каждый из них получил один и тот же вектор длины  $n$ , в котором  $i$ -ый элемент либо содержит численность  $i$ -ой армии (если ее командир лоялен) либо не определен (если командир предатель).
- Соответствующий рекурсивный алгоритм был предложен в 1982 г. (Lamport).
- Проиллюстрируем его для случая  $n=4$  и  $m=1$ . В этом случае алгоритм осуществляется в 4 шага.

# Задача «Византийских генералов»

- 1 шаг. Каждый генерал посыпает всем остальным сообщение, в котором указывает численность своей армии. Лояльные генералы указывают истинное количество, а предатели могут указывать различные числа в разных сообщениях. Генерал-1 указал 1 (одна тысяча воинов), генерал-2 указал 2, генерал-3 указал трем остальным генералам соответственно  $x, y, z$ , а генерал-4 указал 4.
- 2-ой шаг. Каждый формирует свой вектор из имеющейся информации.

Получается:

vect1 (1,2,x,4)

vect2 (1,2,y,4)

vect3 (1,2,3,4)

vect4 (1,2,z,4)

- 3-ий шаг. Каждый посыпает свой вектор всем остальным (генерал-3 посыпает опять произвольные значения).

# Задача «Византийских генералов»

- 3-ий шаг. Каждый посыает свой вектор всем остальным (генерал-3 посыает опять произвольные значения).

Генералы получают следующие вектора:

g1	g2	g3	g4
(1,2,y,4)	(1,2,x,4)	(1,2,x,4)	(1,2,x,4)
(a,b,c,d)	(e,f,g,h)	(1,2,y,4)	(1,2,y,4)
(1,2,z,4)	(1,2,z,4)	(1,2,z,4)	(i,j,k,l)

- 4-ый шаг. Каждый генерал проверяет каждый элемент во всех полученных векторах. Если какое-то значение совпадает по меньшей мере в двух векторах, то оно помещается в результирующий вектор, иначе соответствующий элемент результирующего вектора помечается «неизвестен».
- Все лояльные генералы получают один вектор  $(1,2,\text{«неизвестен»},4)$  - согласие достигнуто.

# Задача «Византийских генералов»

- Если рассмотреть случай  $n=3$  и  $m=1$ , то согласие не будет достигнуто.
- Lamport доказал, что в системе с  $m$  неверно работающими процессорами можно достичь согласия только при наличии  $2m+1$  верно работающих процессоров (более  $2/3$ ).
- Другие авторы доказали, что в распределенной системе с асинхронными процессорами и неограниченными коммуникационными задержками согласие невозможно достичь даже при одном неработающем процессоре (даже если он не подает признаков жизни).
- Применение алгоритма - надежная синхронизация часов.

# Алгоритм надежных неделимых широковещательных рассылок сообщений

- Алгоритм выполняется в две фазы и предполагает наличие в каждом процессоре очередей для запоминания поступающих сообщений. В качестве уникального идентификатора сообщения используется его начальный приоритет - логическое время отправления, значение которого на разных процессорах различно.

## 1-ая фаза

Процесс-отправитель посыпает сообщение группе процессов (список их идентификаторов содержится в сообщении).

При получении этого сообщения процессы:

- Приписывают сообщению приоритет, помечают сообщение как «недоставленное» и буферизуют его. В качестве приоритета используется временная метка (текущее логическое время).
- Информируют отправителя о приписанном сообщению приоритете.

# Алгоритм надежных неделимых широковещательных рассылок сообщений

## 2-ая фаза

При получении ответов от всех адресатов, отправитель:

- Выбирает из всех приписанных сообщению приоритетов максимальный и устанавливает его в качестве окончательного приоритета сообщения.
- Рассылает всем адресатам этот приоритет.

Получив окончательный приоритет, получатель:

- Приписывает сообщению этот приоритет.
- Помечает сообщение как «доставленное».
- Упорядочивает все буферизованные сообщения по возрастанию их приписанных приоритетов.
- Если первое сообщение в очереди отмечено как «доставленное», то оно будет обрабатываться как окончательно полученное.

# Алгоритм надежных неделимых широковещательных рассылок сообщений

Если получатель обнаружит, что он имеет сообщение с пометкой «недоставленное», отправитель которого сломался, то он для завершения выполнения протокола осуществляет следующие два шага в качестве координатора.

1. Опрашивает всех получателей о статусе этого сообщения.

Получатель может ответить одним из трех способов:

- Сообщение отмечено как «недоставленное» и ему приписан такой-то приоритет.
- Сообщение отмечено как «доставленное» и имеет такой-то окончательный приоритет.
- Он не получал это сообщение.

# Алгоритм надежных неделимых широковещательных рассылок сообщений

2. Получив все ответы координатор выполняет следующие действия:

- Если сообщение у какого-то получателя помечено как «доставленное», то его окончательный приоритет рассыпается всем. (Получив это сообщение каждый процесс выполняет шаги фазы 2).
- Иначе координатор заново начинает весь протокол с фазы 1. (Повторная посылка сообщения с одинаковым приоритетом не должна вызывать коллизий).

Необходимо заметить, что алгоритм требует хранения начального и окончательного приоритетов даже для принятых и уже обработанных сообщений.

<https://drive.google.com/drive/folders/192NoY0y8bdVu-ukka3kEGcxuC3Zo1IYH?usp=sharing>

# Суперкомпьютеры петафлопсной производительности

#12 Top500 (June'19, Rmax = 17.590 PFlop/s); 10/12 — 08/19  
200 cabinets; 18,688 nodes; 299,008 cores; 18,688 K20X; 693.6TBytes;  
8.2 MW;



*R. A. Ashraf and C. Engelmann, Analyzing the Impact of System Reliability Events on Applications in the Titan Supercomputer, 8th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS), 2018*

Event log from Reliability, Availability and Serviceability (RAS) System

68k apps have 1+ events (over  $2 \cdot 10^6$ )

95% of apps with 11k+ nodes have 1+ events

91% of apps with 125- nodes have 0 events

85% apps under 30min have 0 events

80% apps above 24h have 1+ events

Nature of first event hitting an application:

Parallel Filesystem	73.7%
Processor Failure	15.7 %
Machine Check Exception	6.5%
GPU Failure	1.5%
OOM / SEGFAULT	1.9%
Interconnect	0.8 %

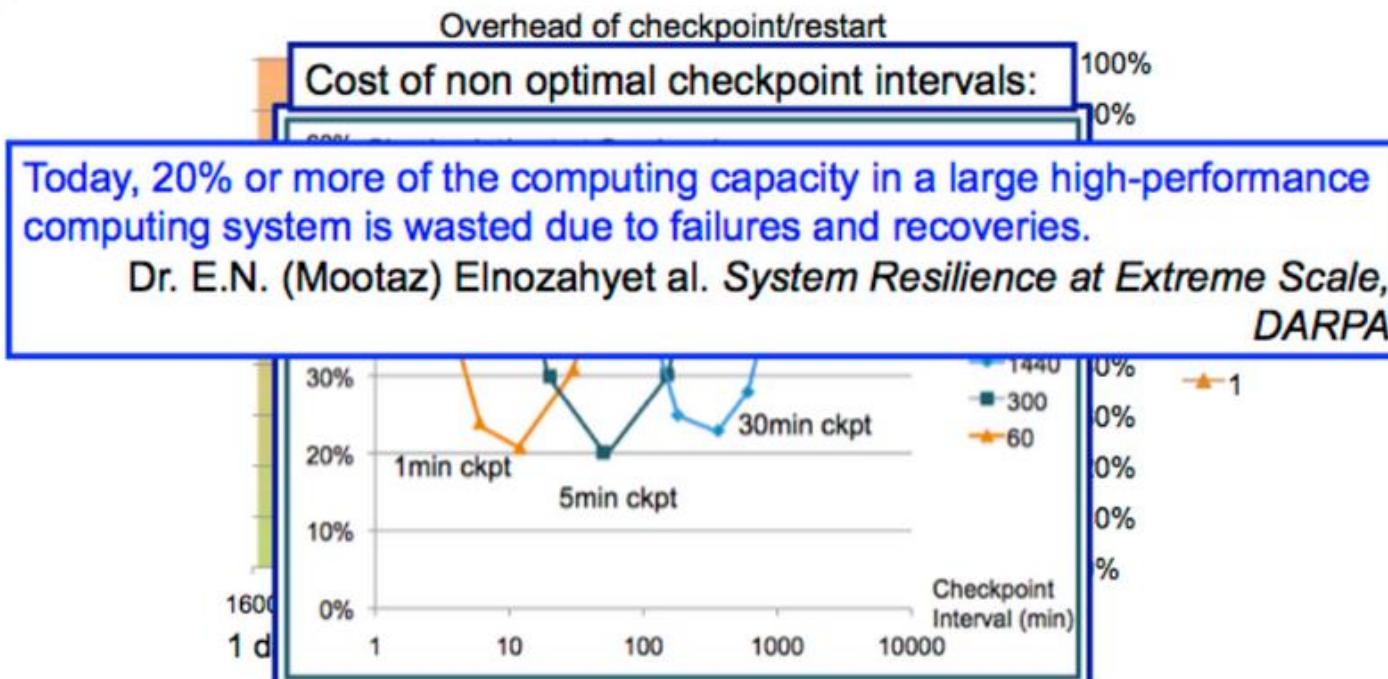
# Суперкомпьютеры петафлопсной производительности

Joint Laboratory for Petascale Computation

## Also an issue at Petascale

RINRIA NCSA

Fault tolerance becomes critical at Petascale (MTTI <= 1day)  
Poor fault tolerance design may lead to huge overhead



# Суперкомпьютеры экзафлопсной производительности

- Pre-exascale machines are hierarchical, many with accelerators
  - Top 10:  $2 \cdot 10^4$  to  $1.5 \cdot 10^5$  nodes
  - Each node equipped with 48 to 256 cores
  - Many of those featuring multiple (up to 6) GPU or other accelerator / node
- Exascale machines
  - Many will feature accelerators
  - Number of nodes should remain in the same order of magnitude
  - Level of parallelism needs to reach 1 billion threads at 1GHz each
- Failure-prone

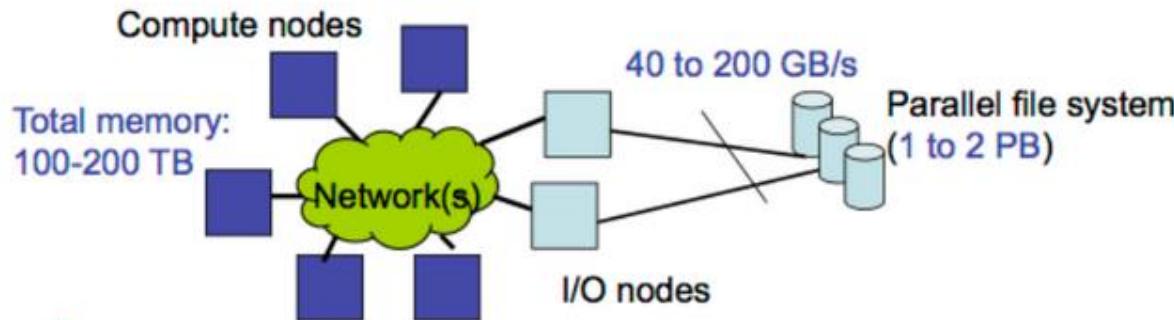
MTBF – one component	1 year	10 years	120 years
MTBF – platform of $10^4$ components	1 min	9 min	4 days
MTBF – platform of $10^5$ components	6 sec	1 min	10h
MTBF – platform of $10^6$ components	< 1 sec	6 sec	1h

More hardware component ⇒ Shorter MTBF (Mean Time Between Failures)

# Контрольные точки

## Classic approach for FT: Checkpoint-Restart

Typical “Balanced Architecture” for PetaScale Computers



Without optimization, Checkpoint-Restart needs about 1h! (~30 minutes each)

Systems	Perf.	Ckpt time	Source
RoadRunner	1PF	~20 min.	Panasas
LLNL BG/L	500 TF	>20 min.	LLNL
LLNL Zeus	11TF	26 min.	LLNL
YYY BG/P	100 TF	~30 min.	YYY

# Обработка ошибок в OpenMP-программах

Директива

**#pragma omp cancel clause[,,] clause ]**

где clause одна из:

- **parallel**
- **sections**
- **for**
- **taskgroup**
- **if (scalar-expression)**

Директива

**#pragma omp cancellation point clause[,,] clause ]**

где clause одна из:

- **parallel**
- **sections**
- **for**
- **taskgroup**

Новая функция системы поддержки:

- ❑ **omp\_get\_cancellation**

Новая переменная окружения:

- ❑ **OMP\_CANCELLATION**

# Обработка ошибок в OpenMP-программах

```
void example() {
    std::exception *ex = NULL;
    #pragma omp parallel shared(ex)
    {
        #pragma omp for schedule (dynamic)
        for (int i = 0; i < N; i++) {
            try {
                causes_an_exception();
            } catch (std::exception *e) {
                #pragma omp atomic write
                ex = e; // still must remember exception for later handling
                #pragma omp cancel for // cancel worksharing construct
            }
        }
        if (ex) { // if an exception has been raised, cancel parallel region
            #pragma omp cancel parallel
        }
    }
    if (ex) { // handle exception stored in ex
    }
}
```

# Поиск в дереве

```
typedef struct binary_tree_s {
    int value;
    struct binary_tree_s *left, *right;
} binary_tree_t;

binary_tree_t *search_tree_parallel (binary_tree_t *tree, int value) {
    binary_tree_t *found = NULL;
#pragma omp parallel shared(found, tree, value)
    {
#pragma omp taskgroup
        {
#pragma omp master
            {
                found = search_tree(tree, value, 0);
            }
        }
    }
    return found;
}
```

# Поиск в дереве

```
binary_tree_t *search_tree(binary_tree_t *tree, int value, int level) {
    binary_tree_t *found = NULL;
    if (tree) {
        if (tree->value == value) {
            found = tree;
        } else {
            #pragma omp task shared(found) if(level < 10)
            {
                binary_tree_t *found_left = NULL;
                found_left = search_tree(tree->left, value, level + 1);
                if (found_left) {
                    #pragma omp atomic write
                    found = found_left;
                    #pragma omp cancel taskgroup
                }
            }
        }
    }
}
```

# Поиск в дереве

```
#pragma omp task shared(found) if(level < 10)
{
    binary_tree_t *found_right = NULL;
    found_right = search_tree(tree->right, value, level + 1);
    if (found_right) {
        #pragma omp atomic write
        found = found_right;
        #pragma omp cancel taskgroup
    }
}
#pragma omp taskwait
}
}
return found;
```

# Сбой во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int main(int argc, char *argv[])
{
    int rank, size, rc, len;
    char errstr[MPI_MAX_ERROR_STRING];
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    rc = MPI_Barrier(MPI_COMM_WORLD);
    MPI_Error_string(rc, errstr, &len);
    printf("Rank %d / %d: Notified of error %s. Stayin' alive!\n", rank, size, errstr);
    MPI_Finalize();
}
```

# Сбой во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int main(int argc, char *argv[])
{
    int rank, size, rc, len;
    char errstr[MPI_MAX_ERROR_STRING];
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    rc = MPI_Barrier(MPI_COMM_WORLD);
    MPI_Error_string(rc, errstr, &len);
    printf("Rank %d / %d: Notified of error %s. Stayin' alive!\n", rank, size, errstr);
    MPI_Finalize();
}
```

# Сбой во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
static void verbose_errhandler(MPI_Comm* comm, int* err, ...) {
    int rank, size, len;
    char errstr[MPI_MAX_ERROR_STRING];
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Error_string( *err, errstr, &len );
    printf("Rank %d / %d: Notified of error %s\n",
        rank, size, errstr);
}
```

# Сбой во время работы MPI-программы

```
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Errhandler errh;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_create_errhandler(verbose_errhandler, &errh);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Rank %d / %d: Stayin' alive!\n", rank, size);
    MPI_Finalize();
}
```

# User Level Failure Mitigation (ULFM)

<https://fault-tolerance.org/>

```
#include <mpi-ext.h>
```

- **MPIX\_ERR\_PROC\_FAILED** when a process failure prevents the completion of an MPI operation.
- **MPIX\_ERR\_PROC\_FAILED\_PENDING** when a potential sender matching a non-blocking wildcard source receive has failed.
- **MPIX\_ERR\_REVOKED** when one of the ranks in the application has invoked the MPI\_Comm\_revocate operation on the communicator.
- **MPIX\_Comm\_revocate(MPI\_Comm comm)** Interrupts any communication pending on the communicator at all ranks.
- **MPIX\_Comm\_shrink(MPI\_Comm comm, MPI\_Comm\* newcomm)** creates a new communicator where dead processes in comm were removed.
- **MPIX\_Comm\_agree(MPI\_Comm comm, int \*flag)** performs a consensus (i.e. fault tolerant allreduce operation) on flag (with the operation bitwise or).
- **MPIX\_Comm\_failure\_get\_acked(MPI\_Comm, MPI\_Group\*)** obtains the group of currently acknowledged failed processes.
- **MPIX\_Comm\_failure\_ack(MPI\_Comm)** acknowledges that the application intends to ignore the effect of currently known failures on wildcard receive completions and agreement return values.

# Сбой во время работы MPI-программы

```
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Errhandler errh;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_create_errhandler(verbose_errhandler, &errh);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
}
```

# Сбой во время работы MPI-программы

```
static void verbose_errhandler(MPI_Comm* pcomm, int* perr, ...) {
    MPI_Comm comm = *pcomm;
    int err = *perr;
    char errstr[MPI_MAX_ERROR_STRING];
    int i, rank, size, nf, len, eclass;
    MPI_Group group_c, group_f;
    int *ranks_gc, *ranks_gf;

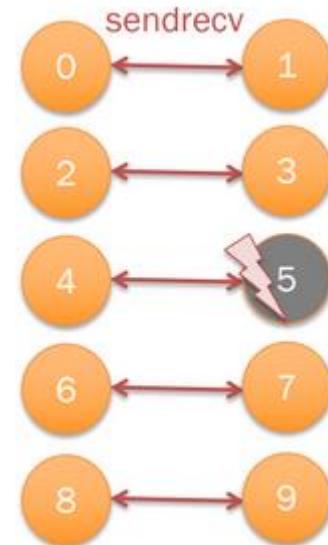
    MPI_Error_class(err, &eclass);
    if( MPIX_ERR_PROC_FAILED != eclass ) {
        MPI_Abort(comm, err);
    }
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);
```

# Сбой во время работы MPI-программы

```
/* We use a combination of 'ack/get_acked' to obtain the list of failed processes. */
MPIX_Comm_failure_ack(comm);
MPIX_Comm_failure_get_acked(comm, &group_f);
MPI_Group_size(group_f, &nf);
MPI_Error_string(err, errstr, &len);
printf("Rank %d / %d: Notified of error %s. %d found dead: { ", rank, size, errstr, nf);
/* We use 'translate_ranks' to obtain the ranks of failed procs in 'comm' communicator */
ranks_gf = (int*)malloc(nf * sizeof(int));
ranks_gc = (int*)malloc(nf * sizeof(int));
MPI_Comm_group(comm, &group_c);
for(i = 0; i < nf; i++)
    ranks_gf[i] = i;
MPI_Group_translate_ranks(group_f, nf, ranks_gf,
                           group_c, ranks_gc);
for(i = 0; i < nf; i++)
    printf("%d ", ranks_gc[i]);
printf("}\n");
free(ranks_gf); free(ranks_gc);
}
```

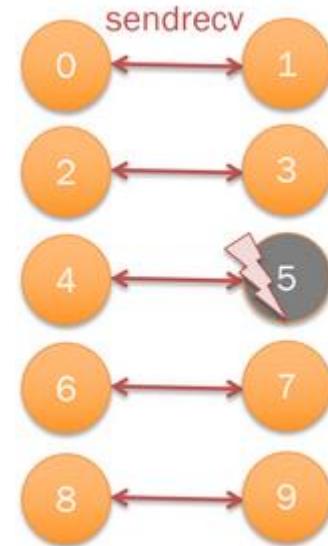
# Сбой во время работы MPI-программы

```
int main(int argc, char *argv[]) {  
    int rank, size, peer;  
    MPI_Errhandler errh;  
    double myvalue, hisvalue=NAN;  
    MPI_Init(NULL, NULL);  
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);  
    MPI_Comm_size(MPI_COMM_WORLD, &size);  
    MPI_Comm_create_errhandler(verbose_errhandler, &errh);  
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);  
    MPI_Barrier(MPI_COMM_WORLD);  
    myvalue = rank/(double)size;  
    if( 0 == rank%2 )  
        peer = ((rank+1)<size)? rank+1: MPI_PROC_NULL;  
    else  
        peer = rank-1;
```



# Сбой во время работы MPI-программы

```
if( rank == (size/2) ) raise(SIGKILL);
/* exchange a value between a pair of two consecutive
 * odd and even ranks; not communicating with anybody
 * else. */
MPI_Sendrecv(&myvalue, 1, MPI_DOUBLE, peer, 1,
             &hisvalue, 1, MPI_DOUBLE, peer, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
if( peer != MPI_PROC_NULL)
    printf("Rank %d / %d: value from %d is %g\n",
          rank, size, peer, hisvalue);
MPI_Finalize();
}
```



# Сбой во время работы MPI-программы

```
if( rank == (size/2) ) raise(SIGKILL);
/* exchange a value between a pair of two consecutive
 * odd and even ranks; not communicating with anybody
 * else. */

MPI_Sendrecv(&myvalue, 1, MPI_DOUBLE, peer, 1,
             &hisvalue, 1, MPI_DOUBLE, peer, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

if( peer != MPI_PROC_NULL)
    printf("Rank %d / %d: value from %d is %g\n",
          rank, size, peer, hisvalue);

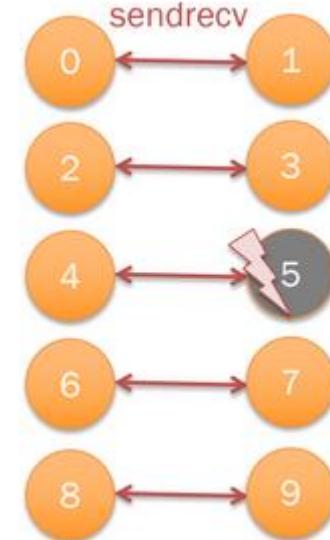
MPI_Finalize();

}
```

```
bash$ $ULFM_PREFIX/bin/mpirun -np 10 03.undisturbed
Rank 0 / 10: value from 1 is 0.1
Rank 1 / 10: value from 0 is 0
Rank 3 / 10: value from 2 is 0.2
Rank 2 / 10: value from 3 is 0.3
Rank 6 / 10: value from 7 is 0.7
Rank 7 / 10: value from 6 is 0.6
Rank 9 / 10: value from 8 is 0.8
Rank 8 / 10: value from 9 is 0.9
Rank 4 / 10: Notified of error MPI_ERR_PROC_FAILED: Process Failure. 1 found dead: { 5 }
Rank 4 / 10: value from 5 is nan
```

Sendrecv between pairs of live processes complete w/o error. Can post more, it will work too!

Sendrecv failed at rank 4 (5 is dead)  
Value not updated!



# Сбой во время работы MPI-программы

```
void error_handler(MPI_Comm *pcomm, int *error_code, ...) {
    int rc, i, myrank, oldrank, num_fails, error_class;
    char error_string[MPI_MAX_ERROR_STRING];
    MPI_Group f_group, w_group;
    int f_group_rank[MAX_SIZE], w_group_rank[MAX_SIZE];
    MPI_Comm communicator = *pcomm, new_comm;
    ...
/* I found a failed process, so I will acknowledge the failure and shrink the
   communicator */
if(0 == myrank) {
    // Get group from current communicator
    MPI_Comm_group(communicator, &w_group);
    // Get group of failed procs
    MPIX_Comm_failure_ack(communicator);
    MPIX_Comm_failure_get_acked(communicator, &f_group);
```

# Сбой во время работы MPI-программы

```
// Get no. of failed procs
MPI_Group_size(f_group, &num_fails);
// Get ranks of failed procs in the original comm
for(i = 0; i < num_fails; i++) f_group_rank[i] = i;
MPI_Group_translate_ranks(f_group, num_fails, f_group_rank,
                           w_group, w_group_rank);
// Stop the workers. Prevent a communicator from being used in the future
MPIX_Comm_revoke(communicator);
}
// Creates a new communitor from an existing communicator while excluding
// failed processes
MPIX_Comm_shrink(communicator, &new_comm);
communicator = new_comm;
MPI_Comm_size(communicator, &myrank);
...
}
```

# Обработка ошибок в MPI-программах

```
static void compute(int how_long);
int main(int argc, char* argv[]) {
    /* Send some large data to make the recv long enough to see something
       interesting */
    int count = 4*1024*1024;
    int* send_buff = malloc(count * sizeof(*send_buff));
    int* recv_buff = malloc(count * sizeof(*recv_buff));
    /* Every process sends a token to the right on a ring (size tokens sent per
       * iteration, one per process per iteration) */
    int left = MPI_PROC_NULL, right = MPI_PROC_NULL;
    int rank = MPI_PROC_NULL, size = 0;
    int tag = 42, iterations = 10, how_long = 5;
    MPI_Request req = MPI_REQUEST_NULL;
    double post_date, wait_date, complete_date;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    left = (rank+size-1)%size; /* left ring neighbor in circular space */
    right = (rank+size+1)%size; /* right ring neighbor in circular space */
```

# Обработка ошибок в MPI-программах

```
**** Done with initialization, main loop now *****
printf("Entering test; computing silently for %d seconds\n", how_long);
for(int i = 0; i < iterations; i++) {
    MPI_Barrier(MPI_COMM_WORLD);
    post_date = MPI_Wtime();
    MPI_Irecv(recv_buff, 512, MPI_INT, left, tag, MPI_COMM_WORLD, &req);
    MPI_Send (send_buff, 512, MPI_INT, right, tag, MPI_COMM_WORLD);
    compute(how_long);
    wait_date = MPI_Wtime();
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    complete_date = MPI_Wtime();
    printf("Worked %g seconds before entering MPI_Wait and spend %g seconds
waiting\n",  wait_date - post_date, complete_date - wait_date);
}
MPI_Finalize();
return 0;
}
```

# Обработка ошибок в MPI-программах

```
**** Done with initialization, main loop now *****
printf("Entering test; computing silently for %d seconds\n", how_long);
for(int i = 0; i < iterations; i++) {
    MPI_Barrier(MPI_COMM_WORLD);
    post_date = MPI_Wtime();
    MPI_Irecv(recv_buff, 512, MPI_INT, left, tag, MPI_COMM_WORLD, &req);
    MPI_Send (send_buff, 512, MPI_INT, right, tag, MPI_COMM_WORLD);
    compute(how_long);
    wait_date = MPI_Wtime();
    MPI_Wait(&req, MPI_STATUS_IGNORE);
    complete_date = MPI_Wtime();
    printf("Worked %g seconds before entering MPI_Wait and spend %g seconds
waiting\n", wait_date - post_date, complete_date - wait_date);
}
MPI_Finalize();      mpirun -mca mpi_ft_detector_timeout 1 -np 10 test
return 0;            Entering test; computing silently for 5 seconds ...
                     [saturn:62859] *** An error occurred in MPI_Send *** reported by process 1
                     [saturn:62859] *** on communicator MPI_COMM_WORLD
                     [saturn:62859] *** MPI_ERR_PROC_FAILED: Process Failure
                     [saturn:62859] *** MPI_ERRORS_ARE_FATAL (processes in this
communicator will now abort, and potentially your MPI job)
```

# Использование дополнительных процессов

**MPI\_Comm\_split(MPI\_Comm comm, int color, int key, MPI\_Comm \*newcomm)**

IN	comm	- родительский коммуникатор;
IN	color	- признак подгруппы;
IN	key	- управление упорядочиванием;
OUT	newcomm	- новый коммуникатор.

Функция расщепляет группу, связанную с родительским коммуникатором, на непересекающиеся подгруппы по одной на каждое значение признака подгруппы color.

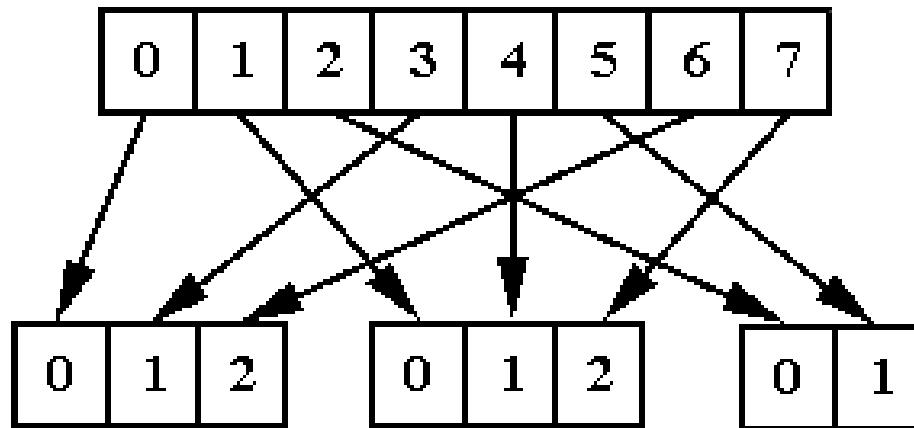
Значение color должно быть неотрицательным. Каждая подгруппа содержит процессы с одним и тем же значением color.

Параметр key управляет упорядочиванием внутри новых групп: меньшему значению key соответствует меньшее значение идентификатора процесса.

В случае равенства параметра key для нескольких процессов упорядочивание выполняется в соответствии с порядком в родительской группе.

# Использование дополнительных процессов

```
MPI_Comm comm, newcomm;  
int myid, color;  
....  
MPI_Comm_rank(comm, &myid);  
color = myid%3;  
MPI_Comm_split(comm, color, myid, &newcomm);
```



# Introduction to MPI I/O

William Gropp [www.cs.illinois.edu/~wgropp](http://www.cs.illinois.edu/~wgropp)

# Checkpoints

---

```
static void save_checkpoint()
{
    if (rank == 0) {
        FILE* file = fopen("gauss.txt",
"w");
        for(int i = 0; i < N; i++) {
            for (int j = 0; j <= N; j++) {
                fprintf(file, "%f", A(i, j));
            }
        }
        fclose(file);
    }
}

float *A;
#define A(i,j) A[(i)*(N+1)+(j)]
```

```
static void load_checkpoint()
{
    FILE* file = fopen("gauss.txt",
"r");
    for(int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            fscanf(file, "%f", &A(i, j));
        }
    }
    fclose(file);
    printf("Proc %d loaded
checkpoint\n", rank);
}
```

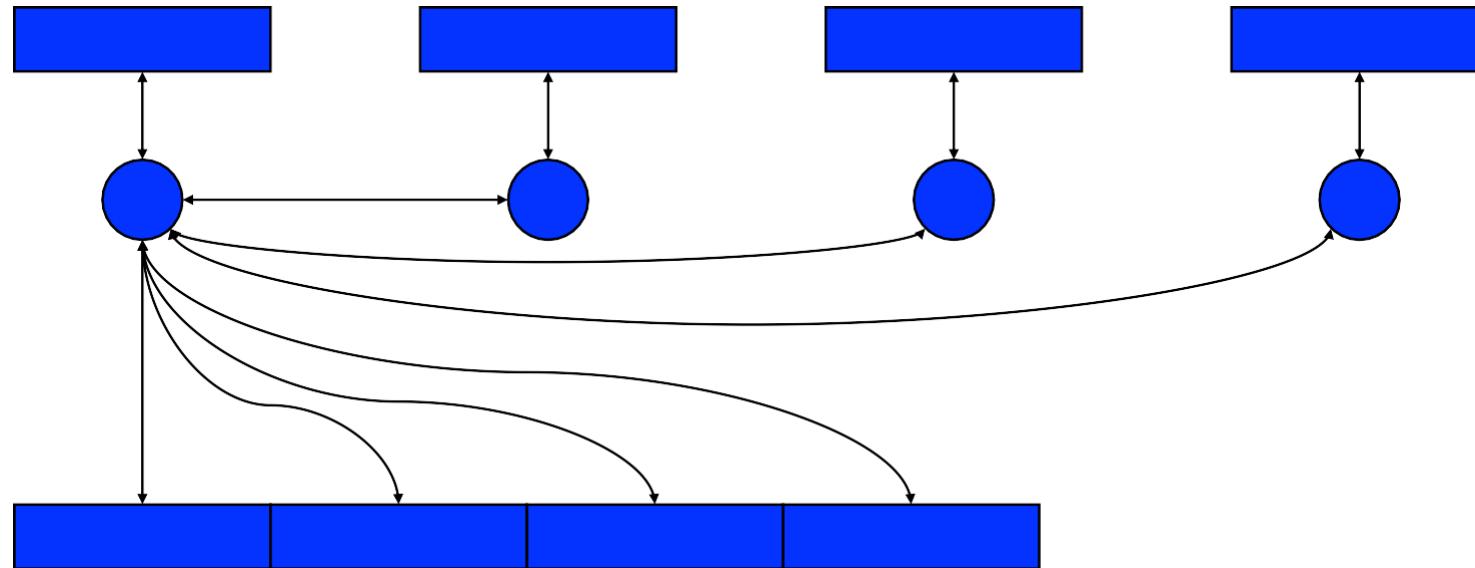
# Parallel I/O in MPI

---

- Why do I/O in MPI?
  - ◆ Why not just POSIX?
    - Parallel performance
    - Single file (instead of one file / process)
- MPI has replacement functions for POSIX I/O
  - ◆ Provides migration path
- Multiple styles of I/O can all be expressed in MPI
  - ◆ Including some that cannot be expressed without MPI

# Non-Parallel I/O

---

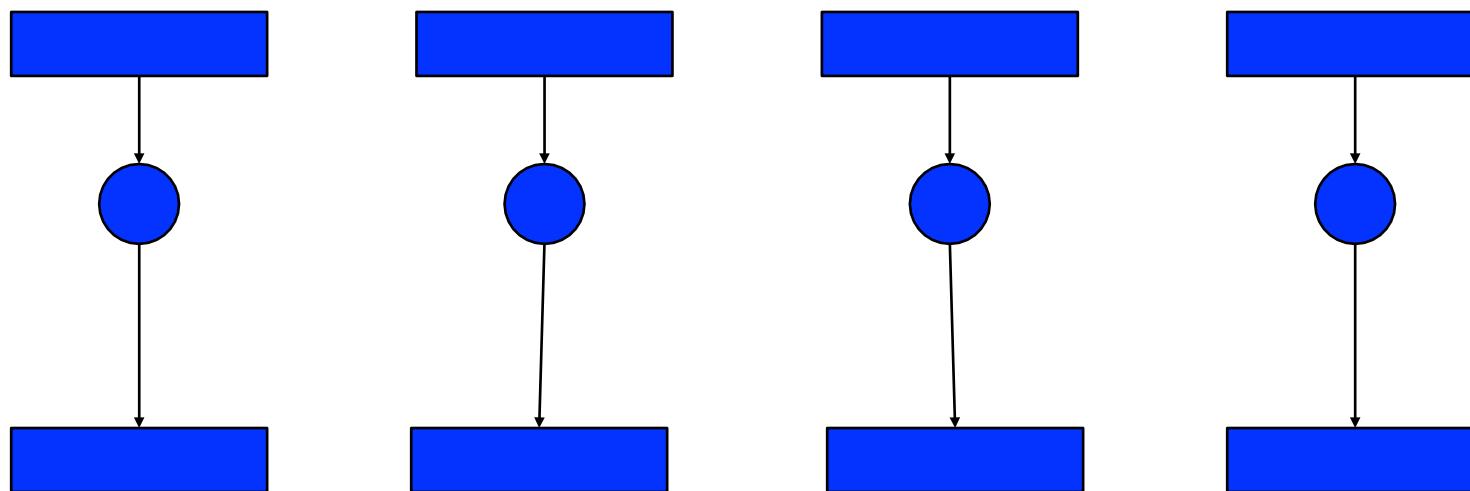


- Non-parallel
- Performance worse than sequential
- Legacy from before application was parallelized
- Either MPI or not

# Independent Parallel I/O

---

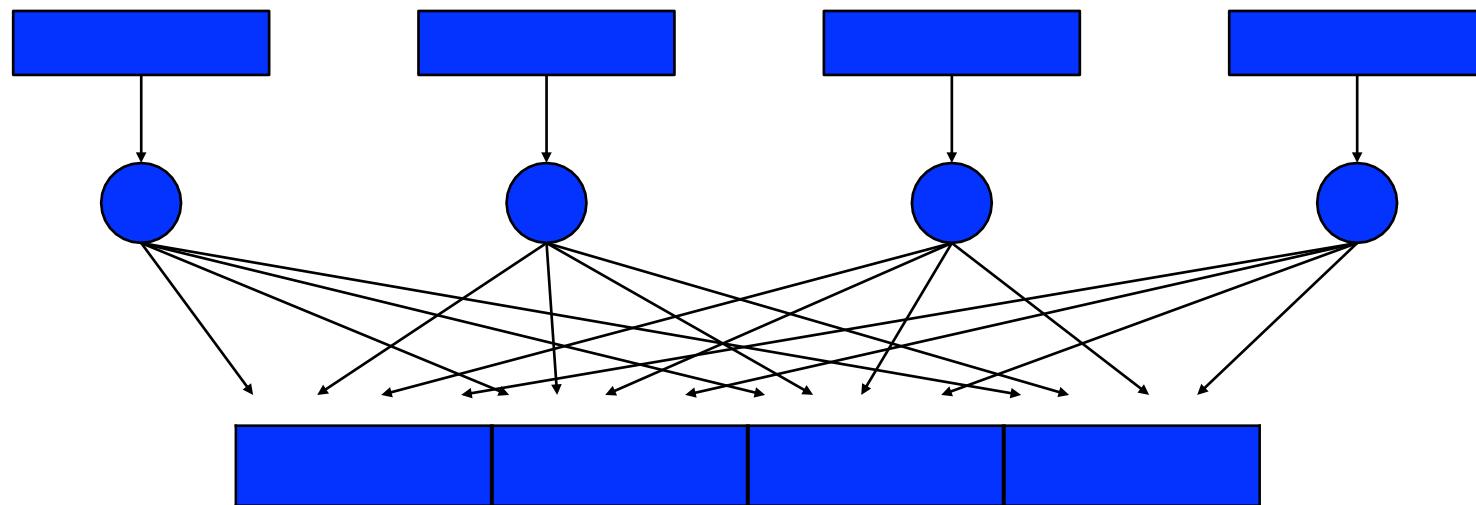
- Each process writes to a separate file



- Pro: parallelism
- Con: lots of small files to manage
- Legacy from before MPI
- MPI or not

# Cooperative Parallel I/O

---



- Parallelism
- Can only be expressed in MPI
- Natural once you get used to it

# Why MPI is a Good Setting for Parallel I/O

---

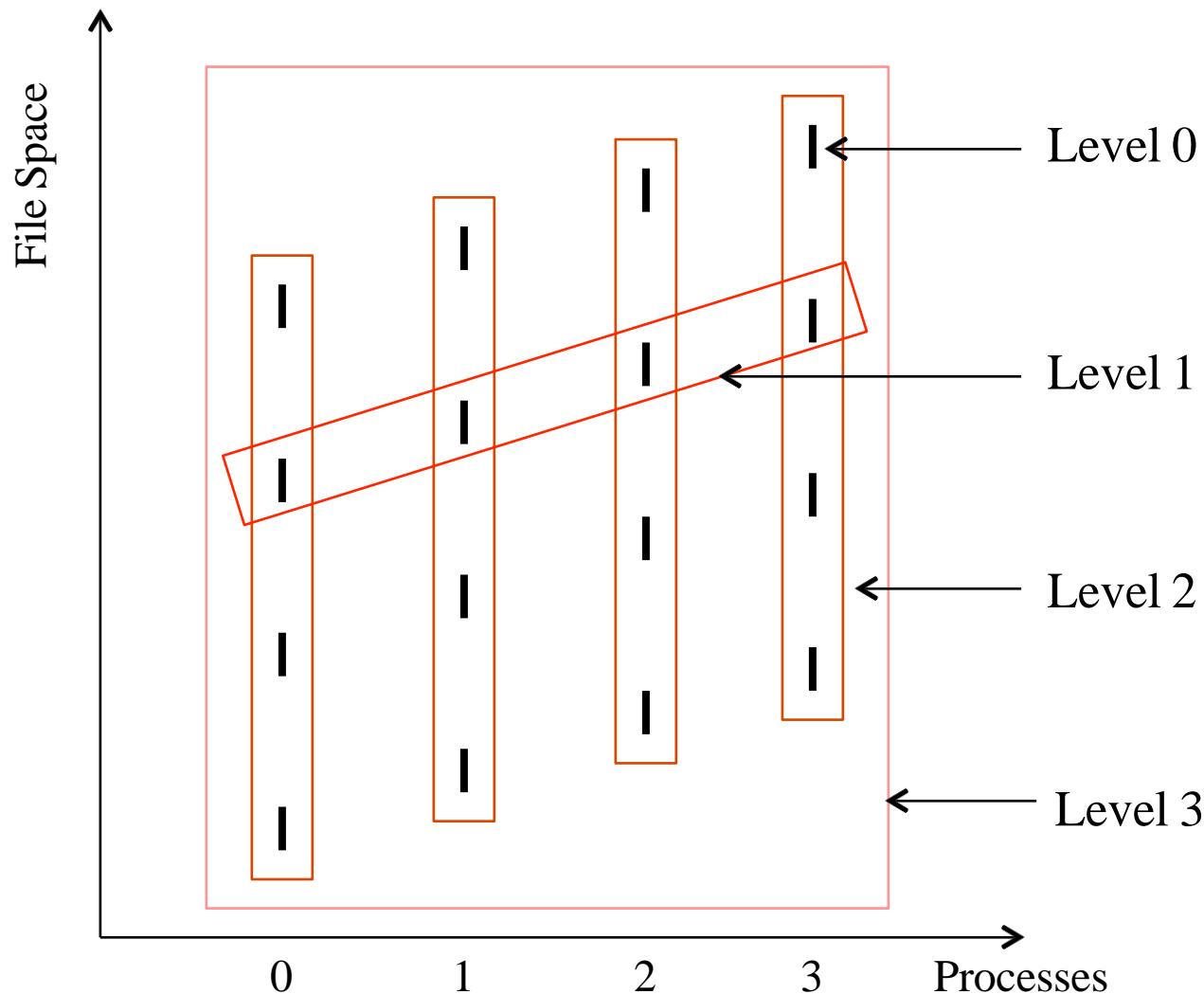
- Writing is like sending and reading is like receiving.
- Any parallel I/O system will need:
  - ◆ collective operations
  - ◆ user-defined datatypes to describe both memory and file layout
  - ◆ communicators to separate application-level message passing from I/O-related message passing
  - ◆ non-blocking operations
- I.e., lots of MPI-like machinery

# What does Parallel I/O Mean?

---

- At the program level:
  - ◆ Concurrent reads or writes from multiple processes to a common file
- At the system level:
  - ◆ A parallel file system and hardware that support such concurrent access

# The Four Levels of Access



---

# Independent I/O with MPI-IO

# The Basics: An Example

---

- Just like POSIX I/O, you need to
  - ◆ Open the file
  - ◆ Read or Write data to the file
  - ◆ Close the file
- In MPI, these steps are almost the same:
  - ◆ Open the file: `MPI_File_open`
  - ◆ Write to the file: `MPI_File_write`
  - ◆ Close the file: `MPI_File_close`

# A Complete Example

---

```
#include <stdio.h> #include "mpi.h"
int main(int argc, char *argv[])
{
    MPI_File fh;
    int buf[1000], rank; MPI_Init(0,0);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_File_open(MPI_COMM_WORLD, "test.out",
                    MPI_MODE_CREATE|MPI_MODE_WRONLY,
                    MPI_INFO_NULL, &fh);
    if (rank == 0)
        MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
    MPI_File_close(&fh); MPI_Finalize();
    return 0;
}
```

# Comments on Example

---

- File Open is collective over the communicator
  - ◆ Will be used to support collective I/O, which we will see is important for performance
  - ◆ Modes similar to Unix open
  - ◆ MPI\_Info provides additional hints for performance
- File Write is independent (hence the test on rank)
  - ◆ Many important variations covered in later slides
- File close is collective; similar in style to MPI\_Comm\_free

# Passing Hints

---

- MPI defines MPI\_Info
- Provides an extensible list of key=value pairs
- Used to package variable, optional types of arguments that may not be standard
  - ◆ Used in IO, Dynamic, and RMA, as well as with communicators

# Example of Hints Display

---

```
PE 0: MPICH/MPIIO environment settings:  
PE 0:  MPICH_MPIIO_HINTS_DISPLAY = 1  
PE 0:  MPICH_MPIIO_HINTS      = NULL  
PE 0:  
MPICH_MPIIO_ABORT_ON_RW_ERROR =  
disable  
PE 0:  MPICH_MPIIO_CB_ALIGN    = 2  
PE 0:  MPIIO hints for ioperf.out.tfaRGQ:  
      cb_buffer_size      = 16777216  
      romio_cb_read        = automatic  
                          = automatic  
      romio_cb_write        = 1  
      cb_nodes             = 2  
      cb_align              = false  
      romio_no_indep_rw     =  
      romio_cb_pfr          = disable  
      romio_cb_fr_types     = aar
```

```
romio_cb_ds_threshold   = 0  
romio_cb_alltoall       = automatic  
ind_rd_buffer_size     = 4194304  
                          = 524288  
ind_wr_buffer_size     = disable  
romio_ds_read           = disable  
romio_ds_write          = 1  
striping_factor         = 1048576  
striping_unit           =  
  
aggregator_placement_stride = -1  
abort_on_rw_error        = disable  
cb_config_list           = *;*
```

# Examples of Hints (used in ROMIO)

---

- `striping_unit`
  - `striping_factor`
  - `cb_buffer_size`
  - `cb_nodes`
  - `ind_rd_buffer_size`
  - `ind_wr_buffer_size`
  - `start_iodevice`
  - `pfs_svr_buf`
  - `direct_read`
  - `direct_write`
- 
- The diagram illustrates the classification of MPI hints into three categories:
- MPI predefined hints:** `striping_unit`, `striping_factor`, `cb_buffer_size`, `cb_nodes`, `ind_rd_buffer_size`, and `ind_wr_buffer_size`. These are grouped by a brace on the right.
  - New Algorithm Parameters:** `start_iodevice`, `pfs_svr_buf`, `direct_read`, and `direct_write`. These are also grouped by a brace on the right.
  - Platform-specific hints:** This category is represented by a brace on the right, covering all hints that do not fall into the first two groups.

# Passing Hints

---

```
MPI_Info info;  
  
MPI_Info_create(&info);  
  
/* no. of I/O devices to be used for file striping */  
MPI_Info_set(info, "striping_factor", "4");  
  
/* the striping unit in bytes */  
MPI_Info_set(info, "striping_unit", "65536");  
  
MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",  
              MPI_MODE_CREATE | MPI_MODE_RDWR, info, &fh);  
  
MPI_Info_free(&info);
```

# Writing to a File

---

- Use `MPI_File_write` or `MPI_File_write_at`
- Use `MPI_MODE_WRONLY` or `MPI_MODE_RDWR` as the flags to `MPI_File_open`
- If the file doesn't exist previously, the flag `MPI_MODE_CREATE` must also be passed to `MPI_File_open`
- We can pass multiple flags by using bitwise-or ‘|’ in C, or addition ‘+’ in Fortran

# Ways to Access a Shared File

---

- `MPI_File_seek`
  - `MPI_File_read`
  - `MPI_File_write`
  - `MPI_File_read_at`
  - `MPI_File_write_at`
  - `MPI_File_read_shared`
  - `MPI_File_write_shared`
- 
- The diagram illustrates the grouping of MPI file access functions. A brace on the right side groups the first three functions (`MPI_File_seek`, `MPI_File_read`, `MPI_File_write`) under the label "like Unix I/O". Another brace groups the next two functions (`MPI_File_read_at`, `MPI_File_write_at`) under the label "combine seek and I/O for thread safety". A third brace groups the last two functions (`MPI_File_read_shared`, `MPI_File_write_shared`) under the label "use shared file pointer".

# Using Explicit Offsets

---

```
#include "mpi.h"
MPI_Status status;
MPI_File fh;
MPI_Offset offset;

MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
              MPI_MODE_RDONLY, MPI_INFO_NULL, &fh)
nints = FILESIZE / (nprocs*INTSIZE);
offset = rank * nints * INTSIZE;
MPI_File_read_at(fh, offset, buf, nints, MPI_INT,
                  &status);
MPI_Get_count(&status, MPI_INT, &count);
printf("process %d read %d
ints\n", rank, count);
MPI_File_close(&fh);
```

# Why Use Independent I/O?

---

- Sometimes the synchronization of collective calls is not natural
- Sometimes the overhead of collective calls outweighs their benefits
  - ◆ Example: very small I/O during header reads

# Noncontiguous I/O in File

---

- Each process describes the part of the file for which it is responsible
  - ◆ This is the “file view”
  - ◆ Described in MPI with an offset (useful for headers) and an MPI\_Datatype
- Only the part of the file described by the file view is visible to the process; reads and writes access these locations
- This provides an efficient way to perform *noncontiguous accesses*

# Noncontiguous Accesses

---

- Common in parallel applications
- Example: distributed arrays stored in files
- A big advantage of MPI I/O over Unix I/O is the ability to specify noncontiguous accesses in memory **and** file within a single function call by using derived datatypes
  - ◆ POSIX only supports non-contiguous in file, and only with IOVs
- Allows implementation to optimize the access
- Collective I/O combined with noncontiguous accesses yields the highest performance

# File Views

---

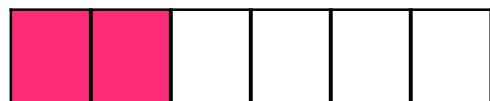
- Specified by a triplet (*displacement*, *etype*, and *filetype*) passed to **`MPI_File_set_view`**
- *displacement* = number of bytes to be skipped from the start of the file
  - ◆ e.g., to skip a file header
- *etype* = basic unit of data access (can be any basic or derived datatype)
- *filetype* = specifies which portion of the file is visible to the process

# A Simple Noncontiguous File View Example

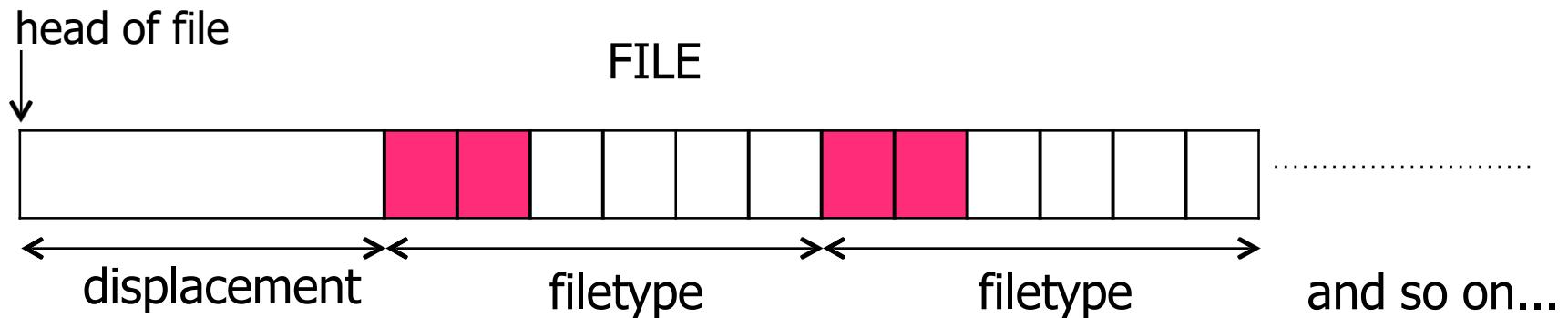
---



etype = MPI\_INT



filetype = two MPI\_INTs followed by  
a gap of four MPI\_INTs



# Noncontiguous FileView Code

---

```
MPI_Aint lb, extent;
MPI_Datatype etype, filetype, contig;
MPI_Offset disp;

MPI_Type_contiguous(2, MPI_INT, &contig);
lb = 0;
extent = 6 * sizeof(int);
MPI_Type_create_resized(contig, lb, extent, &filetype);
MPI_Type_commit(&filetype);
disp = 5 * sizeof(int);
etype = MPI_INT;

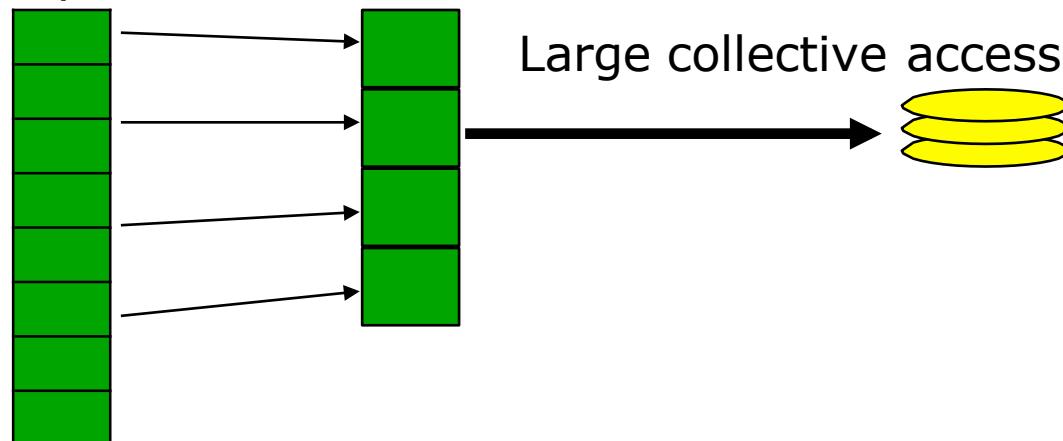
MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile",
    MPI_MODE_CREATE | MPI_MODE_RDWR, MPI_INFO_NULL, &fh);
MPI_File_set_view(fh, disp, etype, filetype, "native",
    MPI_INFO_NULL);
MPI_File_write(fh, buf, 1000, MPI_INT, MPI_STATUS_IGNORE);
```

# Collective I/O and MPI

---

- A critical optimization in parallel I/O
- All processes (in the communicator) must call the collective I/O function
- Allows communication of “big picture” to file system
  - ◆ Framework for I/O optimizations at the MPI-IO layer
- Basic idea: build large blocks, so that reads/writes in I/O system will be large
  - ◆ Requests from different processes may be merged together
  - ◆ Particularly effective when the accesses of different processes are noncontiguous and interleaved

Small individual requests



# Collective I/O Functions

---

- **MPI\_File\_write\_at\_all**, etc.
  - ◆ \_all indicates that all processes in the group specified by the communicator passed to **MPI\_File\_open** will call this function
  - ◆ \_at indicates that the position in the file is specified as part of the call; this provides thread-safety and clearer code than using a separate “seek” call
- Each process specifies only its own access information — the argument list is the same as for the non-collective functions

# The Other Collective I/O Calls

---

- `MPI_File_seek`
  - `MPI_File_read_all`
  - `MPI_File_write_all`
  - `MPI_File_read_at_all`
  - `MPI_File_write_at_all`
  - `MPI_File_read_ordered`
  - `MPI_File_write_ordered`
- 
- The diagram illustrates the grouping of MPI file I/O calls. It features three curly braces on the right side of the list. The first brace groups the first four calls (`MPI_File_seek`, `MPI_File_read_all`, `MPI_File_write_all`, and `MPI_File_read_at_all`) under the label "like Unix I/O". The second brace groups the next two calls (`MPI_File_write_at_all` and `MPI_File_read_ordered`) under the label "combine seek and I/O for thread safety". The third brace groups the last two calls (`MPI_File_write_ordered`) under the label "use shared file pointer".
- like Unix I/O
- combine seek and I/O  
for thread safety
- use shared file pointer

# Using the Right MPI-IO Function

---

- Any application has a particular “I/O access pattern” based on its I/O needs
- The same access pattern can be presented to the I/O system in different ways depending on what I/O functions are used and how
- We classify the different ways of expressing I/O access patterns in MPI-IO into four levels:  
level 0 – level 3
- We demonstrate how the user’s choice of level affects performance

# Example: Distributed Array Access

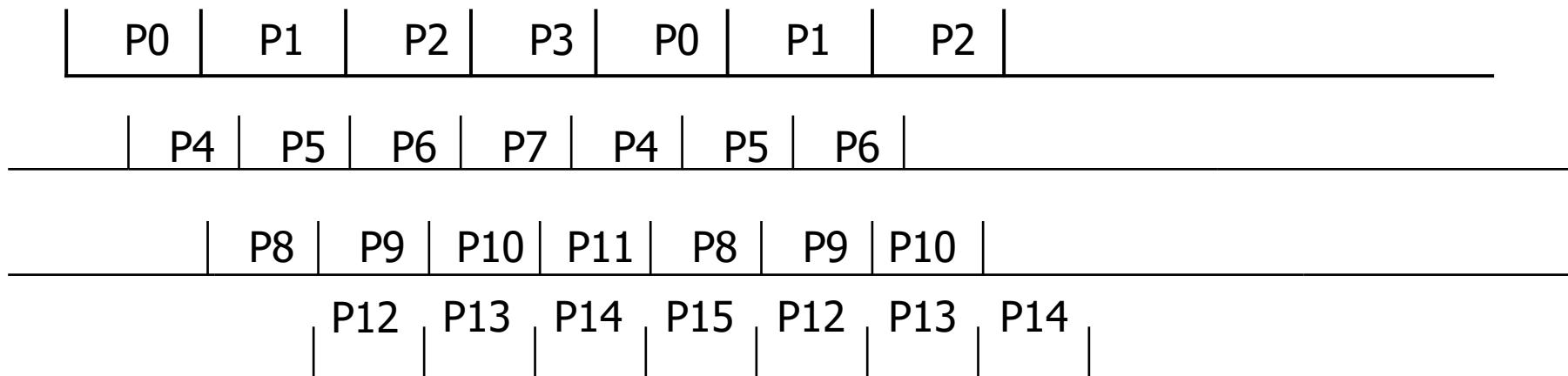
---

Large array distributed among 16 processes

P0	P1	P2	P3
P4	P5	P6	P7
P8	P9	P10	P11
P12	P13	P14	P15

Each square represents a subarray in the memory of a single process

Access Pattern in the file



# Level-0 Access

---

- Each process makes one independent read request for each row in the local array (as in Unix)

```
MPI_File_open(..., file, ..., &fh) ;  
for (i=0; i<n_local_rows; i++) {  
    MPI_File_seek(fh, ...) ;  
    MPI_File_read(fh, &(A[i][0]), ...) ;  
}  
MPI_File_close(&fh) ;
```

# Level-1 Access

---

- Similar to level 0, but each process uses collective I/O functions

```
MPI_File_open(MPI_COMM_WORLD, file, ...,
              &fh);
for (i=0; i<n_local_rows;   i++) {
    MPI_File_seek(fh, ...);
    MPI_File_read_all(fh, &(A[i][0]), ...);
}
MPI_File_close(&fh);
```

# Level-2 Access

---

- Each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent I/O functions

```
MPI_Type_create_subarray(...,  
                      &subarray, ...);  
  
MPI_Type_commit(&subarray);  
  
MPI_File_open(..., file, ..., &fh);  
  
MPI_File_set_view(fh, ..., subarray, ...);  
  
MPI_File_read(fh, A, ...);  
  
MPI_File_close(&fh);
```

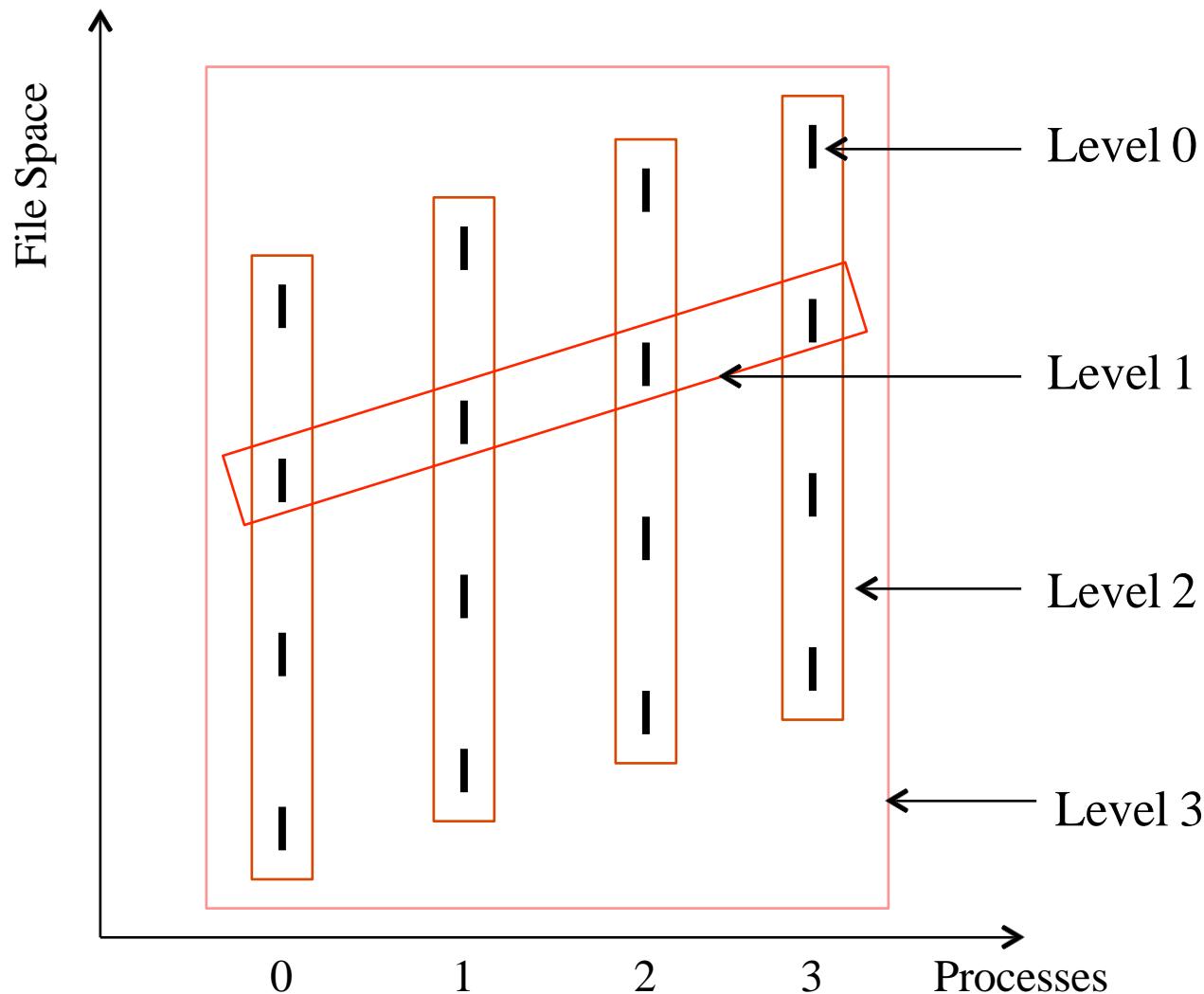
# Level-3 Access

---

- Similar to level 2, except that each process uses collective I/O functions

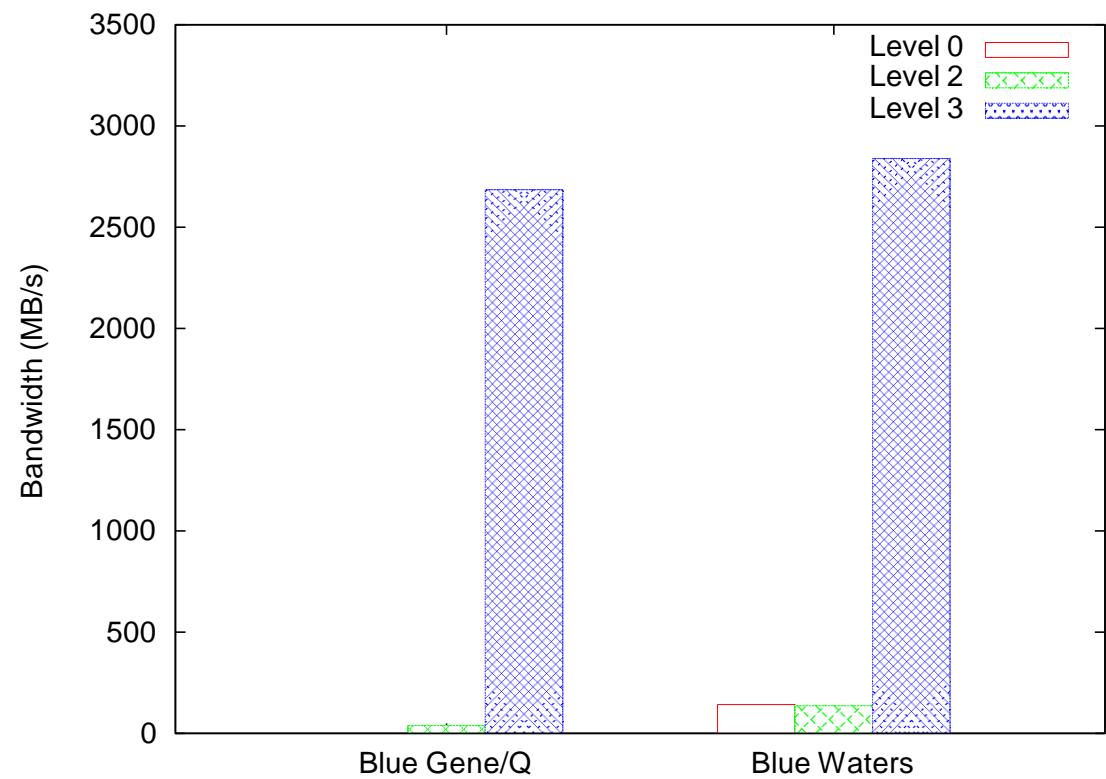
```
MPI_Type_create_subarray(...,  
                      &subarray, ...);  
  
MPI_Type_commit(&subarray);  
  
MPI_File_open(MPI_COMM_WORLD, file, ...,  
              &fh);  
  
MPI_File_set_view(fh    ..., ...  
,           subarray, ...);  
  
MPI_File_read_all(fh  A, ...);  
  
'  
  
MPI_File_close(&fh); 22
```

# The Four Levels of Access



# Collective I/O Can Provide Far Higher Performance

- Write performance for a 3D array output in canonical order on 2 supercomputers, using 256 processes (1 process / core)
- Level 0 (independent I/O from each process for each contiguous block of memory) too slow on BG/Q
- Total BW is still low because relatively few nodes in use (16 for Blue Waters = ~180MB/sec/node)

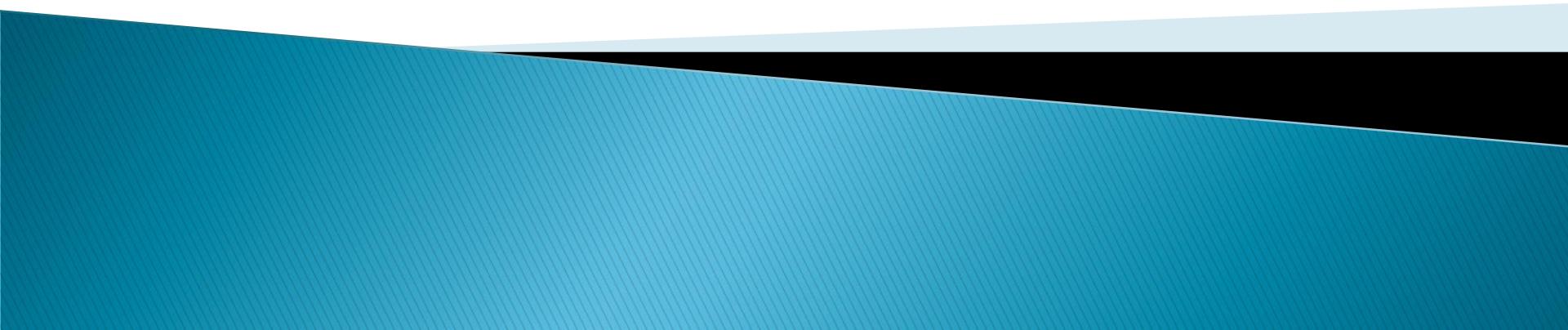


# Summary

---

- Key issues that I/O must address
  - ◆ High latency of devices
    - Nonblocking I/O; cooperative I/O
  - ◆ I/O inefficient if transfers are not both large and aligned with device blocks
    - Collective I/O; datatypes and file views
  - ◆ Data consistency to other users
    - POSIX is far too strong (primary reason parallel file systems have reliability problems)
    - “Big Data” file systems are weak (eventual consistency; tolerate differences)
    - MPI is precise and provides high performance; consistency points guided by users

# Обзор технологии параллельного программирования MPI



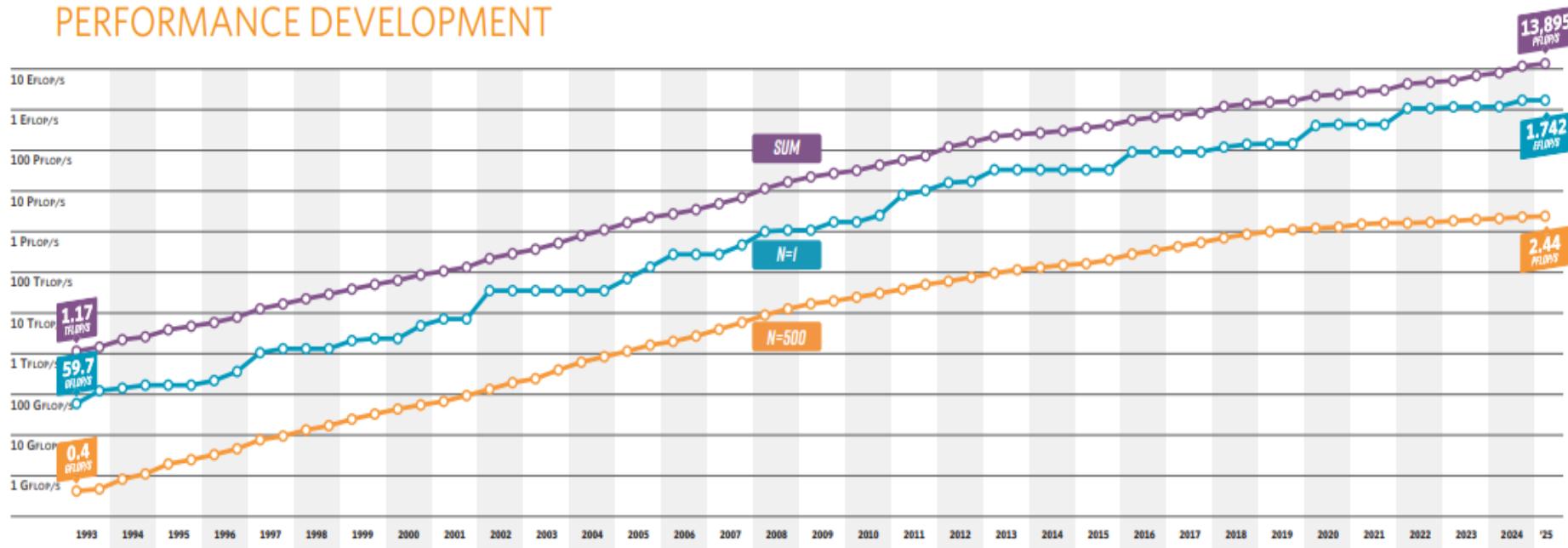
# План лекции

- ▶ Стандарт MPI
- ▶ Основные понятия
- ▶ Блокирующие двухточечные обмены
- ▶ Двухточечные обмены с буферизацией,  
другие типы двухточечных обменов
- ▶ Коллективные операции

JUNE 2025

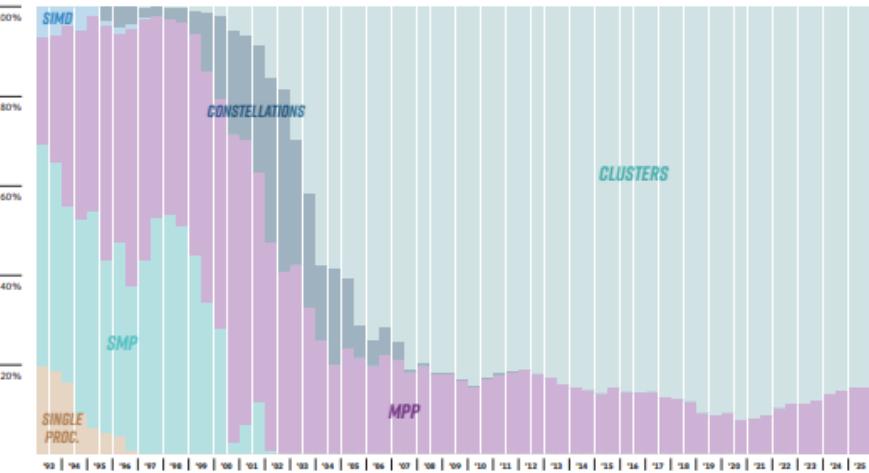
			SITE	COUNTRY	CORES	R <sub>MAX</sub> PFLOP/S	POWER MW
1	<b>El Capitan</b> HPE Cray EX255a, AMD 4th Gen EPYC (24C 1.8GHz), AMD Instinct MI300A, Slingshot-11		DOE/NNSA/LLNL	USA	9,988,224	1,742.0	28.9
2	<b>Frontier</b> HPE Cray EX235a, AMD Opt 3rd Gen EPYC (64C 2GHz), AMD Instinct MI250X, Slingshot-11		DOE/SC/ORNL	USA	9,066,176	1,353.0	24.8
3	<b>Aurora</b> HPE Cray EX Intel Exascale Compute Blade, Xeon CPU Max 9470 (52C 2.4GHz), Intel Data Center GPU Max, Slingshot-11		DOE/SC/ANL	USA	8,159,232	1,012.0	38.7
4	<b>Jupiter Booster</b> BullSequana XH3000, GH Superchip (72C 3GHz), NVIDIA GH200, Quad-Rail NVIDIA InfiniBand		EuroHPC/FZJ	Germany	4,801,344	793.4	13.1
5	<b>Eagle</b> Microsoft NDv5, Xeon Platinum 8480C (48C 2GHz), NVIDIA H100, NVIDIA Infiniband NDR		Microsoft Azure	USA	2,073,600	561.2	

## PERFORMANCE DEVELOPMENT

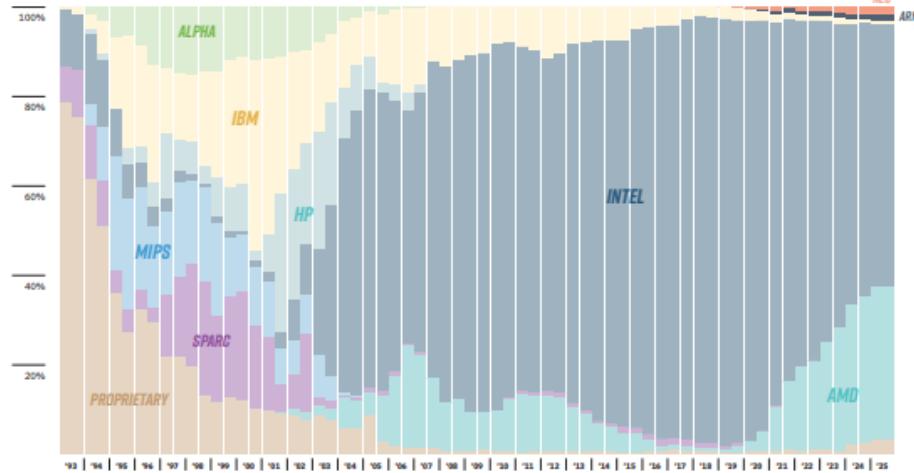


Рейтинг ТОР-500

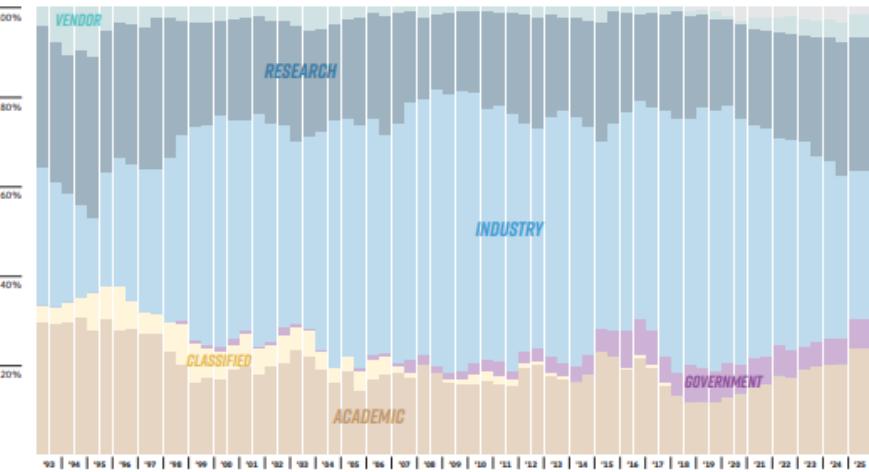
## ARCHITECTURES



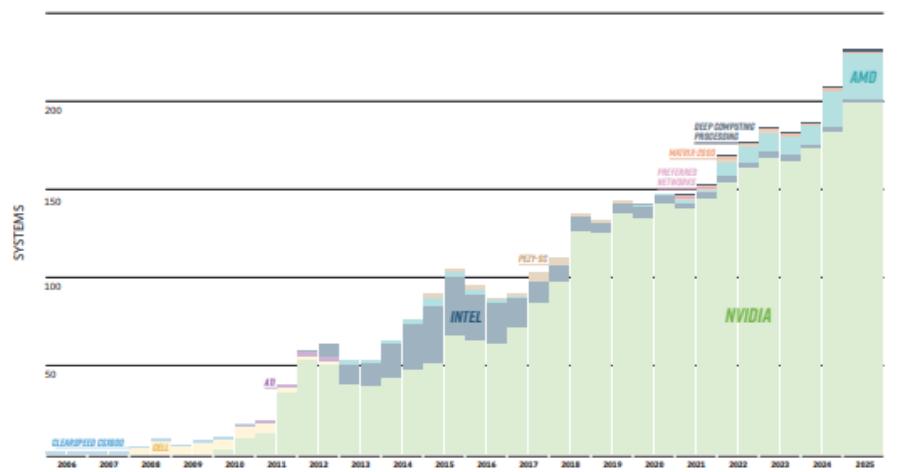
## CHIP TECHNOLOGY



## INSTALLATION TYPE



## ACCELERATORS/CO-PROCESSORS

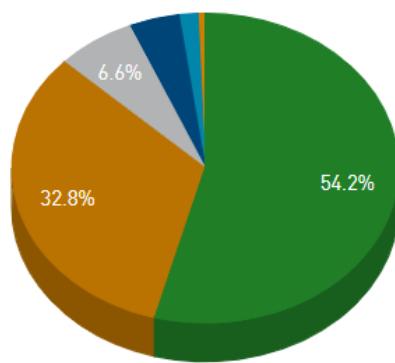


# Рейтинг ТОР-500

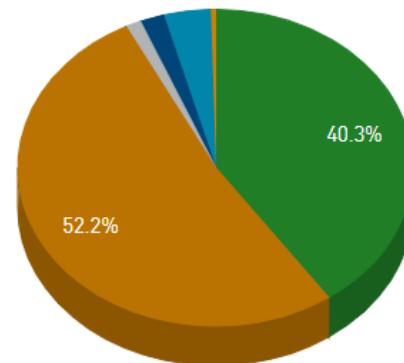
# Рейтинг TOP-500

	Interconnect Family	Count	System Share (%)	Rmax (GFlops)	Rpeak (GFlops)	Cores
1	Infiniband	271	54.2	5,576,009,220	7,976,776,531	43,262,416
2	Gigabit Ethernet	164	32.8	7,224,400,280	11,272,702,991	61,736,268
3	Omnipath	33	6.6	175,546,078	251,438,199	3,833,596
4	Custom Interconnect	21	4.2	271,785,808	388,802,938	19,515,736
5	Proprietary Network	8	1.6	530,839,400	641,021,027	9,105,408
6	Ethernet	3	0.6	58,950,700	82,416,960	253,536

Interconnect Family System Share



Interconnect Family Performance Share



- Infiniband
- Gigabit Ethernet
- Omnipath
- Custom Interconnect
- Proprietary Network
- Ethernet

# Рейтинг TOP-500

Interconnect	Count	System Share (%)	Rmax (GFlops)	Rpeak (GFlops)	Cores
1 100G Ethernet	62	12.4	221,213,310	457,897,700	7,071,568
2 Infiniband HDR	50	10	473,150,370	691,519,450	5,208,184
3 Slingshot-11	46	9.2	6,659,344,270	10,246,456,291	47,618,816
4 Infiniband NDR200	25	5	522,881,390	730,930,629	3,261,328
5 Intel Omni-Path	24	4.8	122,810,558	187,196,188	2,907,500
6 Infiniband NDR400	23	4.6	1,076,749,000	1,647,946,150	3,669,120
7 Mellanox HDR Infiniband	22	4.4	230,856,930	321,339,965	3,192,516
8 Infiniband NDR	20	4	400,313,340	561,143,520	1,968,064
9 25G Ethernet	20	4	63,884,200	124,834,260	1,541,012
10 Infiniband EDR	19	3.8	87,505,060	136,178,904	1,865,404
11 InfiniBand HDR100	19	3.8	87,148,240	126,993,772	2,534,800
12 10G Ethernet	17	3.4	46,660,430	95,658,110	1,383,080
13 Aries interconnect	17	3.4	82,123,614	107,530,618	2,512,728
14 Slingshot-10	13	2.6	118,989,070	157,206,580	3,295,248
15 Infiniband FDR	11	2.2	38,544,310	56,737,170	1,400,296
16 Infiniband HDR200	11	2.2	122,334,010	187,948,610	1,836,992
17 Infiniband	11	2.2	125,581,600	180,389,890	1,086,112
18 Mellanox InfiniBand EDR	9	1.8	60,981,550	95,175,939	1,010,728
19 InfiniBand HDR100	8	1.6	50,897,750	95,729,740	1,384,776
20 Tofu interconnect D	8	1.6	530,839,400	641,021,027	9,105,408

# Infiniband

	Year	Signaling rate (Gbit/s)	Throughput (Gbit/s)				Adapter latency (μs)
			1x	4x	8x	12x	
SDR	2001, 2003	2.5	2	8	16	24	5
DDR	2005	5	4	16	32	48	2.5
QDR	2007	10	8	32	64	96	1.3
FDR10	2011	10.3125	10	40	80	120	0.7
FDR	2011	14.0625	13.64	54.54	109.08	163.64	0.7
EDR	2014	25.78125	25	100	200	300	0.5
HDR	2018	53.125	50	200	400	600	<0.6
NDR	2022	106.25	100	400	800	1200	?
XDR	2024	200	200	800	1600	2400	[to be determined]
GDR	TBA	400	400	1600	3200	4800	

# MPI

- ▶ MPI 1.1 Standard разрабатывался 92–94
- ▶ MPI 2.0 – 95–97
- ▶ MPI 2.1 – сентябрь 2008 г.
- ▶ MPI 3.0 – сентябрь 2012 г.
- ▶ MPI 3.1 – июнь 2015 г.
- ▶ MPI 4.0 – июнь 2021 г.
- ▶ MPI 4.1 – ноябрь 2023 г.
- ▶ MPI 5.0 – июнь 2025 г.

Более 450 процедур

- ▶ Стандарты  
<http://www.mpi-forum.org/>  
<https://computing.llnl.gov/tutorials/mpi/>
- ▶ Описание функций  
<http://www-unix.mcs.anl.gov/mpi/www/>

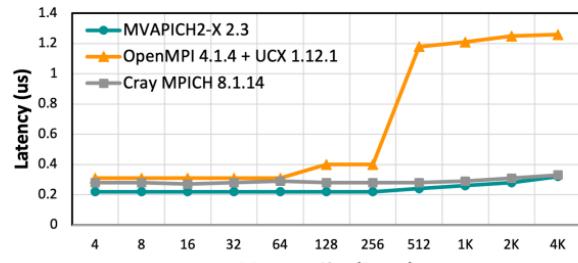
# Цель MPI

- ▶ Основная цель:
  - Обеспечение переносимости исходных кодов
  - Эффективная реализация
- ▶ Кроме того:
  - Большая функциональность
  - Поддержка неоднородных параллельных архитектур

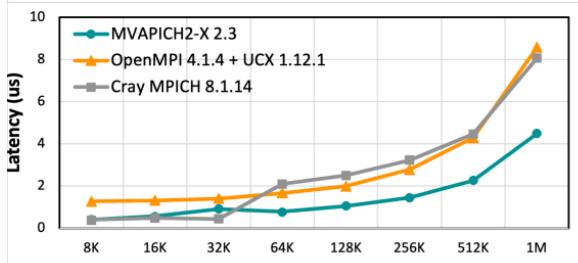
# Реализации MPI

- ▶ MPICH
- ▶ LAM/MPI
- ▶ Mvapich
- ▶ OpenMPI
- ▶ Коммерческие реализации Intel, IBM, Cray и др.

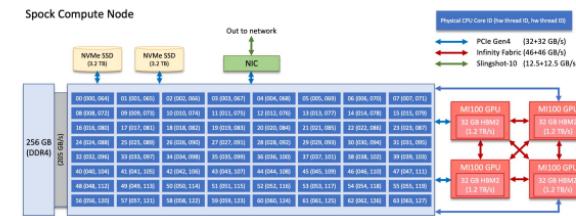
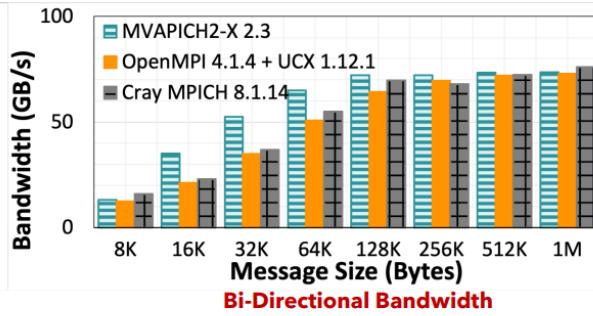
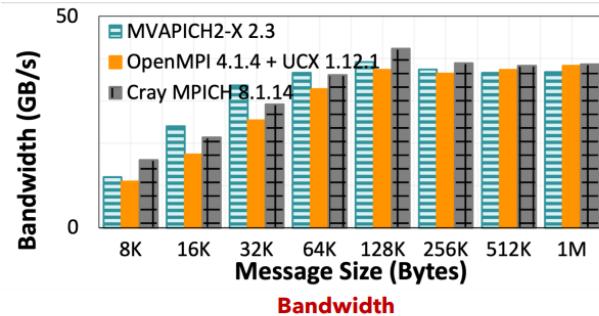
# Point-to-Point Performance - Intra-Node CPU



Latency (small messages)



Latency (large messages)



## Peak Bandwidth:

- MVAPICH2-X **39.2 GB/s**
- OpenMPI+UCX **38.2 GB/s**
- CrayMPICH **42 GB/s**

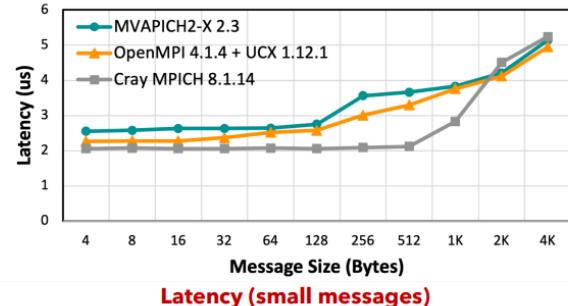
## Latency at 4 Bytes:

- MVAPICH2-X **0.22 us**
- OpenMPI+UCX **0.31 us**
- CrayMPICH **0.27 us**

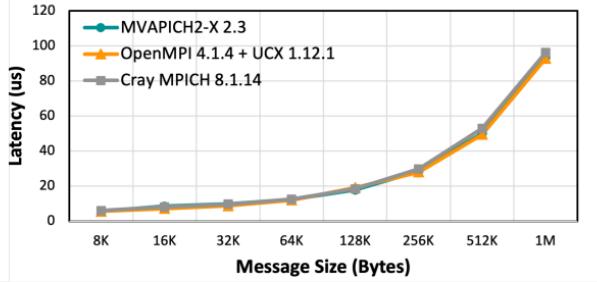
## AMD Epyc Rome CPUs on Spock System

Reference: High Performance MPI over the Slingshot Interconnect: Early Experiences K. Khorassani, C. Chen, B. Ramesh, A. Shafi, H. Subramoni, D. Panda Practice and Experience in Advanced Research Computing, Jul 2022.

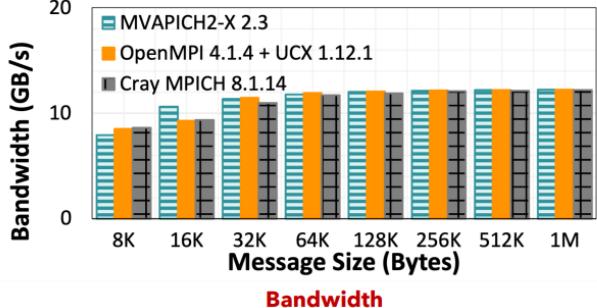
# Point-to-Point Performance - Inter-Node CPU



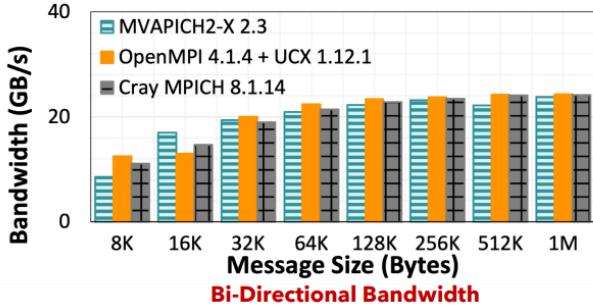
Latency (small messages)



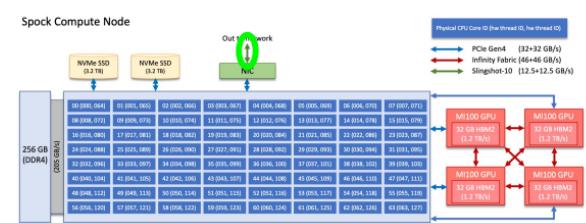
Latency (large messages)



Bandwidth



Bi-Directional Bandwidth



Slingshot-10 Interconnect for over network communication (12.5+12.5 GB/s)

## Peak Bandwidth:

- MVAPICH2-X **122.4 MB/s**
- OpenMPI+UCX **122.4 MB/s**
- CrayMPICH **122.4 MB/s**

## Latency at 4 Bytes:

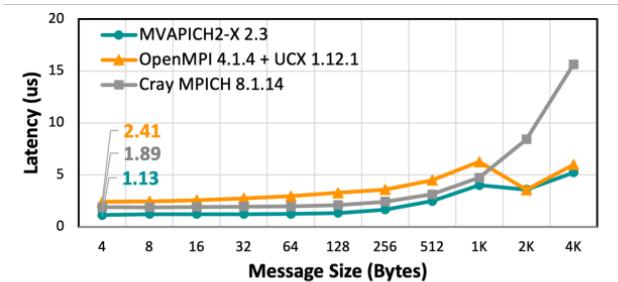
- MVAPICH2-X **2.55 us**
- OpenMPI+UCX **2.27 us**
- CrayMPICH **2.07 us**

## AMD Epyc Rome CPUs on Spock System

Reference: High Performance MPI over the Slingshot Interconnect: Early Experiences K. Khorassani, C. Chen, B. Ramesh, A. Shafi, H. Subramoni, D. Panda Practice and Experience in Advanced Research Computing, Jul 2022.

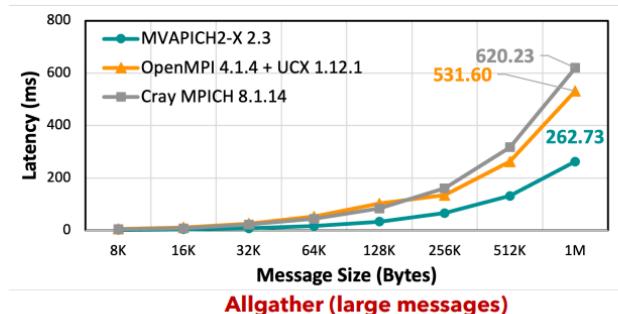
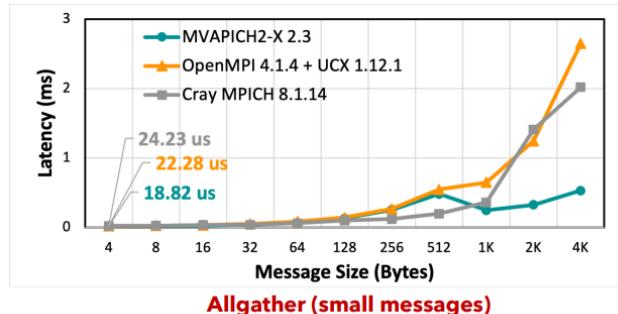
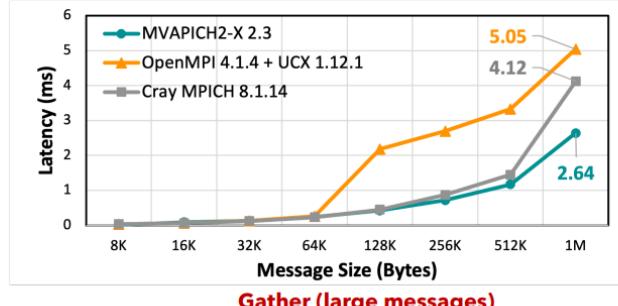
# Collectives Performance - CPU

## GATHER



## ALLGATHER

Gather (small messages)

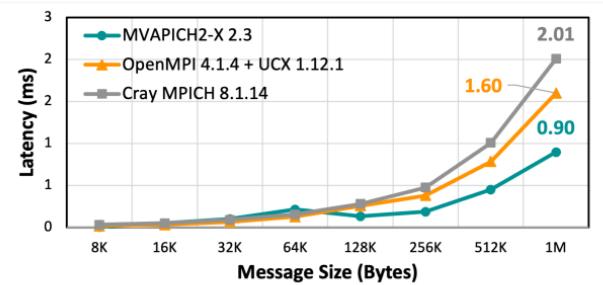
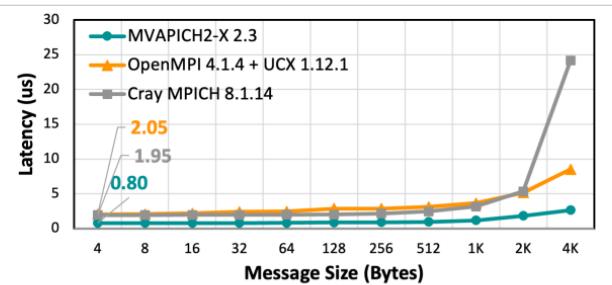


256 CPUs - 4 Nodes & 64 PPN on Spock System

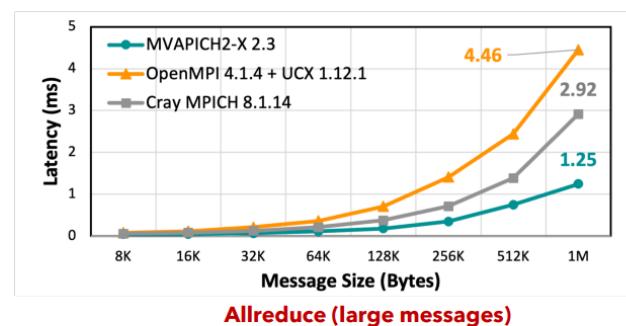
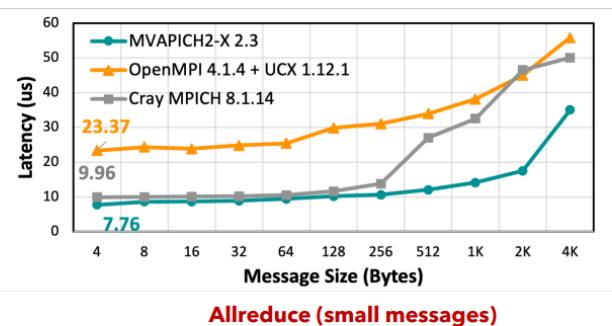
Reference: High Performance MPI over the Slingshot Interconnect: Early Experiences K. Khorassani, C. Chen, B. Ramesh, A. Shafi, H. Subramoni, D. Panda Practice and Experience in Advanced Research Computing, Jul 2022.

# Collectives Performance - CPU

## REDUCE



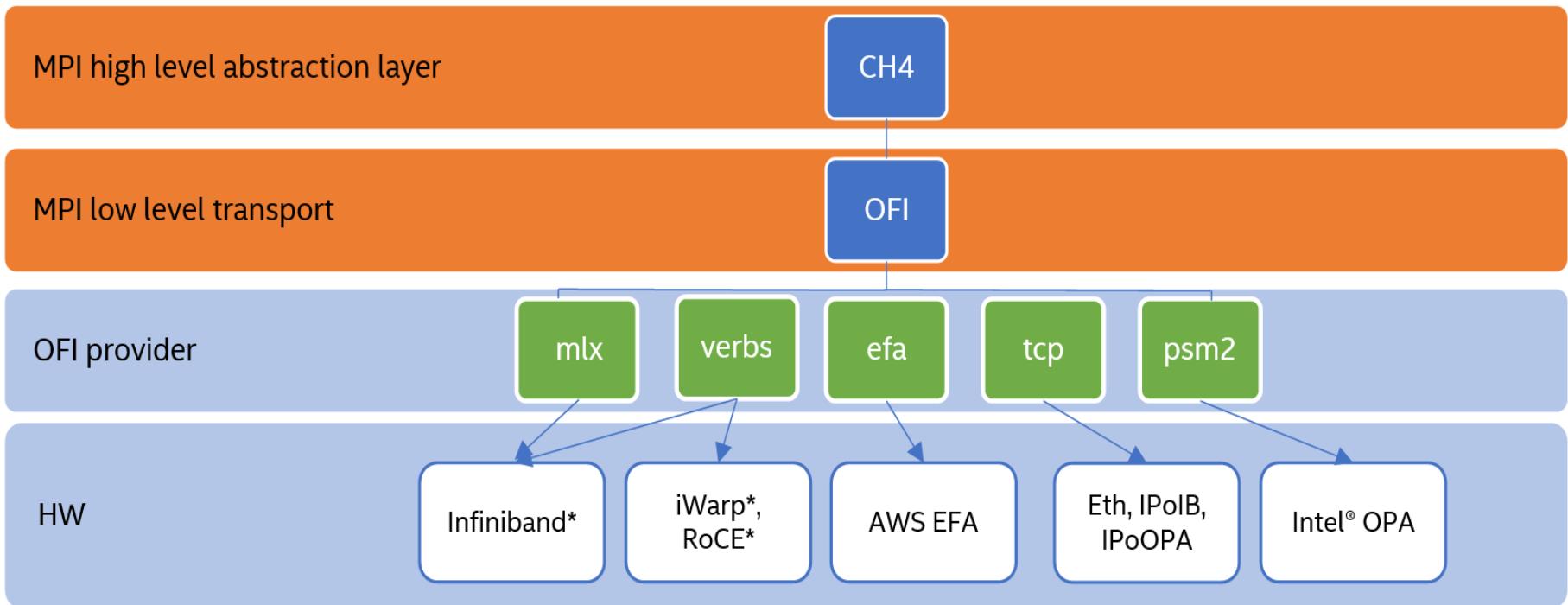
## ALLREDUCE



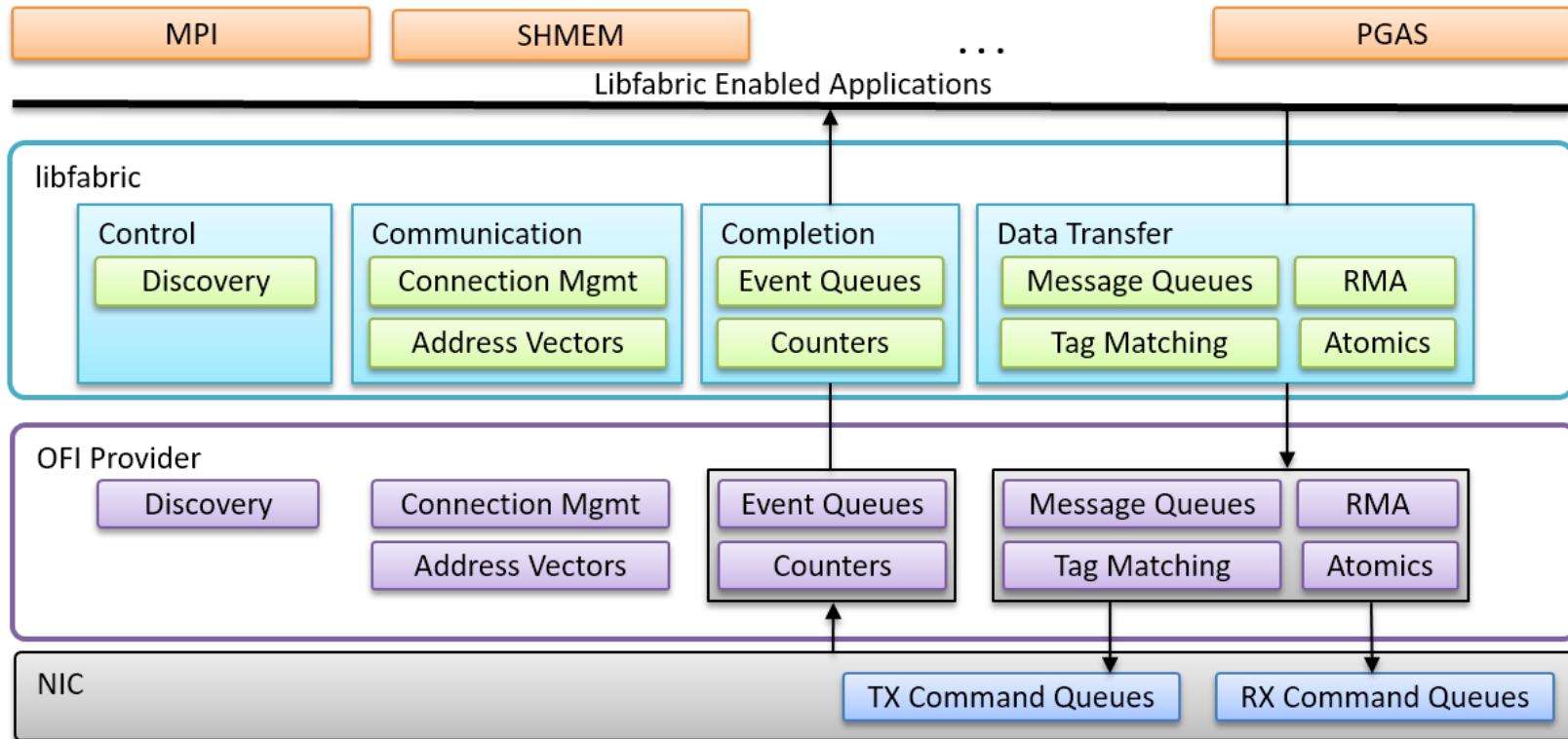
256 CPUs - 4 Nodes & 64 PPN on Spock System

Reference: High Performance MPI over the Slingshot Interconnect: Early Experiences K. Khorassani, C. Chen, B. Ramesh, A. Shafi, H. Subramoni, D. Panda Practice and Experience in Advanced Research Computing, Jul 2022.

# Реализация MPI



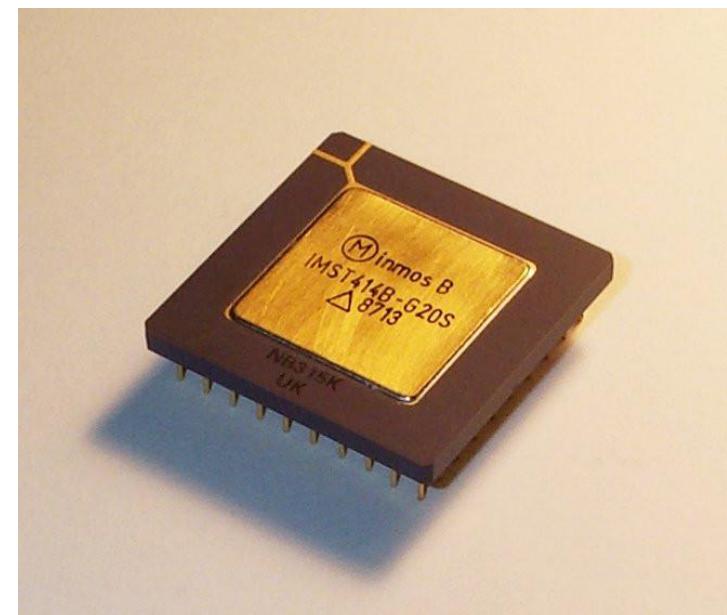
# Реализация MPI. Libfabric OpenFabrics



# Транспьютеры



Четыре высокоскоростных двунаправленных канала связи, которые позволяют напрямую связываться с другими транспьютерами со скоростью передачи данных 5, 10 или 20 Мбит/с.



# Модель MPI

- ▶ Параллельная программа состоит из процессов, процессы могут быть многопоточными.
- ▶ MPI реализует передачу сообщений между процессами.
- ▶ Межпроцессное взаимодействие предполагает:
  - синхронизацию
  - перемещение данных из адресного пространства одного процесса в адресное пространство другого процесса.

# Основные понятия

- ▶ Процессы объединяются в группы.
- ▶ Каждое сообщение посылается в рамках некоторого контекста и должно быть получено в том же контексте.
- ▶ Группа и контекст вместе определяют коммуникатор.
- ▶ Процесс идентифицируется своим номером в группе, ассоциированной с коммуникатором.
- ▶ Коммуникатор, содержащий все начальные процессы, называется MPI\_COMM\_WORLD.

# Понятие коммуникатора MPI

- ▶ Управляющий объект, представляющий группу процессов, которые могут взаимодействовать друг с другом
- ▶ Все обращения к MPI функциям содержат коммуникатор, как параметр
- ▶ Наиболее часто используемый коммуникатор MPI\_COMM\_WORLD
- ▶ Определяется при вызове MPI\_Init
- ▶ Содержит ВСЕ процессы программы

# Типы данных MPI

- ▶ Данные в сообщении описываются тройкой: (address, count, datatype), где datatype определяется рекурсивно как:
  - Предопределенный базовый тип, соответствующий типу данных в базовом языке (например, MPI\_INT, MPI\_DOUBLE\_PRECISION)
  - Непрерывный массив MPI типов
  - Векторный тип
  - Индексированный тип
  - Произвольные структуры
- ▶ MPI включает функции для построения пользовательских типов данных, например, типа данных, описывающих пары (int, float).

# Базовые типы данных

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

# Понятие тэга

- ▶ Сообщение сопровождается определяемым пользователем признаком для идентификации принимаемого сообщения.
- ▶ Теги сообщений у отправителя и получателя должны быть согласованы.
- ▶ Можно указать в качестве значения тэга константу MPI\_ANY\_TAG.
- ▶ Некоторые не-MPI системы передачи сообщений называют тэг типом сообщения.
- ▶ MPI вводит понятие тэга, чтобы не путать это понятие с типом данных MPI.

# Формат MPI-функций

```
error = MPI_Xxxxx(parameter,...);  
MPI_Xxxxx(parameter,...);
```

- ▶ Возвращаемое значение – код ошибки. Определяется константой MPI\_SUCCESS

```
int error;  
.....  
error = MPI_Init(&argc, &argv));  
if (error != MPI_SUCCESS)  
{  
    fprintf (stderr, “ MPI_Init error \n”);  
    return 1;  
}
```

# Выполнение MPI-программы

- ▶ При запуске указываем число требуемых процессоров `pr` и название программы `mpirun -np 3 prog`
- ▶ На выделенных для расчета узлах запускается `pr` копий указанной программы
- ▶ Каждая копия программы получает два значения:
  - `pr`
  - `rank` из диапазона  $[0 \dots np-1]$
- ▶ Любые две копии программы могут непосредственно обмениваться данными с помощью функций передачи сообщений

# C: MPI helloworld.c

```
#include <mpi.h>
main(int argc, char **argv)
{
    int numtasks, rank;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &
numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello World from process %d of %d\n",
rank, numtasks);
    MPI_Finalize();
}
```

# ФУНКЦИИ ОПРЕДЕЛЕНИЯ СРЕДЫ

- ▶ **int MPI\_Init(int \*argc, char \*\*\*argv)**  
должна первым вызовом, вызывается только один раз
- ▶ **int MPI\_Comm\_size(MPI\_Comm comm, int \*size)**  
число процессов в коммуникаторе
- ▶ **int MPI\_Comm\_rank(MPI\_Comm comm, int \*rank)**  
номер процесса в коммуникаторе (нумерация с 0)
- ▶ **int MPI\_Finalize()**  
завершает работу процесса
- ▶ **int MPI\_Abort (MPI\_Comm comm, int\*errorcode)**  
завершает работу программы

# Функции определения среды

- ▶ **int MPI\_Initialized(int \*flag)**

В аргументе **flag** возвращает 1, если вызвана после процедуры **MPI\_Init**, и 0 в противном случае.

- ▶ **int MPI\_Finalized(int \*flag)**

В аргументе **flag** возвращает 1, если вызвана после процедуры **MPI\_Finalize**, и 0 в противном случае.

Эти процедуры можно вызвать до **MPI\_Init** и после **MPI\_Finalize**.

- ▶ **int MPI\_Get\_processor\_name(char \*name, int \*len)**

Возвращает в строке **name** имя узла, на котором запущен вызвавший процесс. В переменной **len** возвращается количество символов в имени, не превышающее константы **MPI\_MAX\_PROCESSOR\_NAME**.

# Функции работы с таймером

- ▶ **double MPI\_Wtime(void)**

Возвращает для каждого вызвавшего процесса астрономическое время в секундах (вещественное число двойной точности), прошедшее с некоторого момента в прошлом. Момент времени, используемый в качестве точки отсчёта, не будет изменён за время существования процесса.

- ▶ **double MPI\_Wtick(void)**

Возвращает разрешение таймера в секундах.

# Использование таймера

```
#include <stdio.h>
#include "mpi.h"
#define NTIMES 1000
int main(int argc, char **argv)
{
    double time_start, time_finish, tick;
    int rank, i;
    int len;
    char *name;
    name = (char*)malloc(MPI_MAX_PROCESSOR_NAME*sizeof(char));
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(name, &len);
    tick = MPI_Wtick();
    time_start = MPI_Wtime();
    for (i = 0; i<NTIMES; i++) time_finish = MPI_Wtime();
    printf ("node %s, process %d: Tick= %lf, time= %lf\n",
           name, rank, tick, (time_finish-time_start)/NTIMES);
    MPI_Finalize();
}
```

# Информация о статусе сообщения

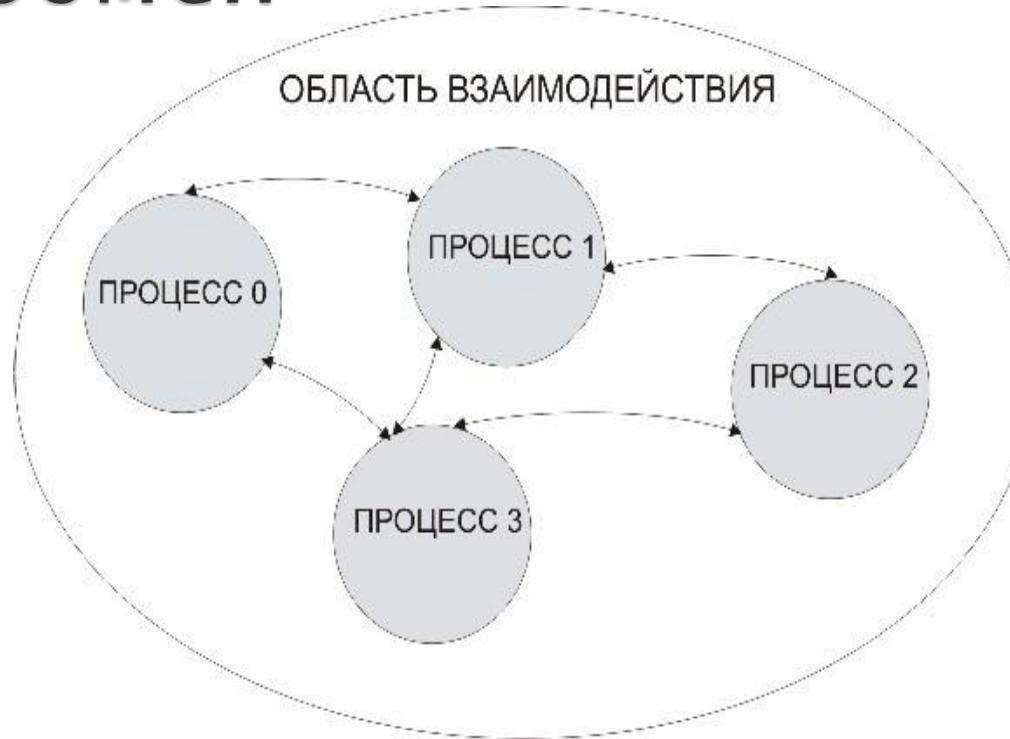
Содержит:

- ▶ Source: status.MPI\_SOURCE
- ▶ Tag: status.MPI\_TAG
- ▶ Код ошибки: status.MPI\_ERROR
- ▶ Count: MPI\_Get\_count

# Двухточечный (point-to-point, p2p) обмен

- ▶ В двухточечном обмене участвуют только два процесса, процесс-отправитель и процесс-получатель (источник сообщения и адресат).
- ▶ Двухточечные обмены используются для организации локальных и неструктурированных коммуникаций.

# Двухточечный (point-to-point, p2p) обмен

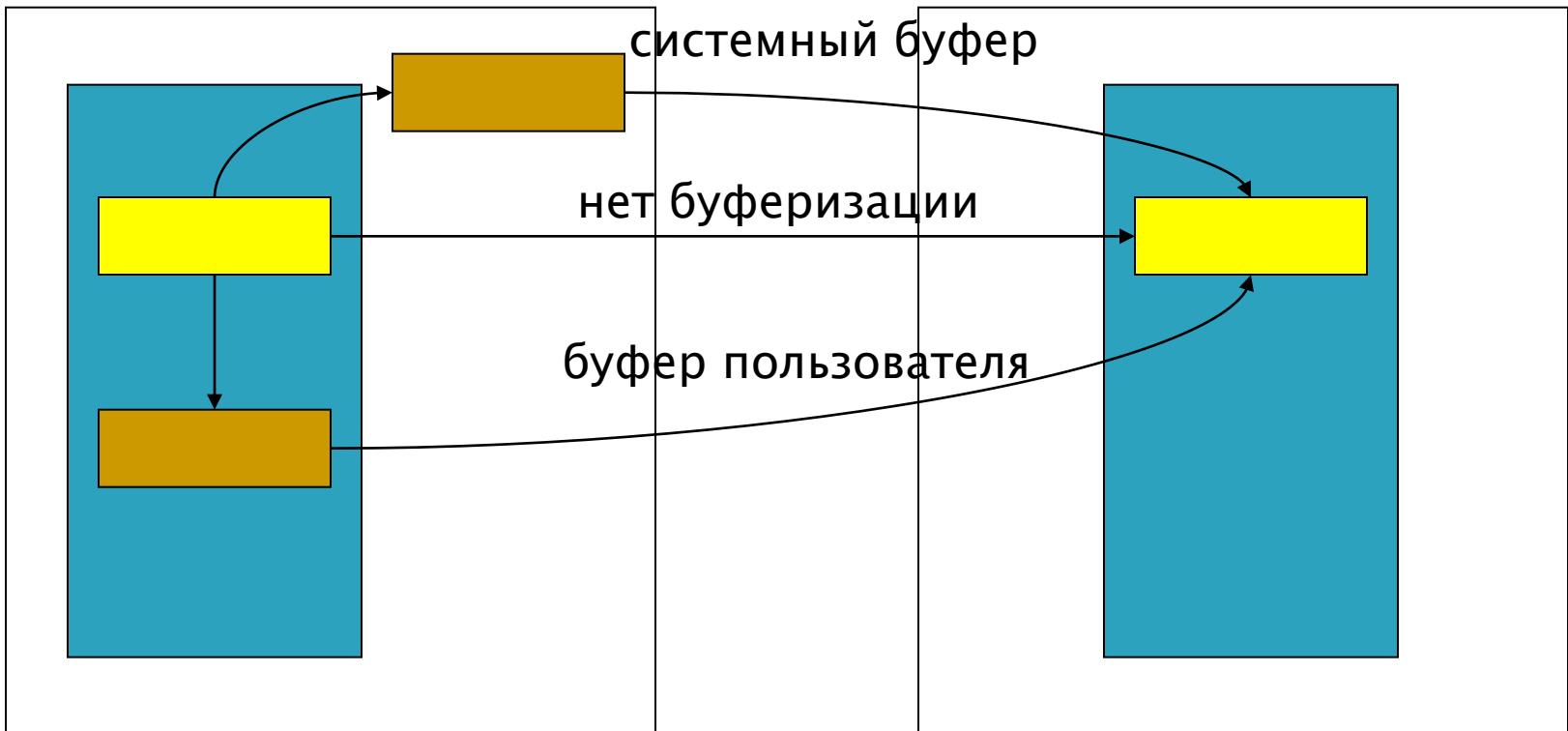


- ▶ Двухточечный обмен возможен только между процессами, принадлежащими одной области взаимодействия (одному коммуникатору).

# Условия успешного взаимодействия точка-точка

- ▶ Отправитель должен указать правильный rank получателя
- ▶ Получатель должен указать верный rank отправителя
- ▶ Одинаковый коммуникатор
- ▶ Тэги должны соответствовать друг другу
- ▶ Буфер у процесса-получателя должен быть достаточного объема

# Выполнение двухточечных обменов



# Разновидности двухточечного обмена

- ▶ *blokiрующие* прием/передача, которые приостанавливают выполнение процесса на время приема или передачи сообщения;
- ▶ *неблокирующие* прием/передача, при которых выполнение процесса продолжается в фоновом режиме, а программа в нужный момент может запросить подтверждение завершения приема сообщения.

# Двухточечный (point-to-point, p2p) обмен

- ▶ Правильно организованный двухточечный обмен сообщениями должен исключать возможность блокировки или некорректной работы параллельной MPI-программы.
- ▶ Примеры ошибок в организации двухточечных обменов:
  - ❑ выполняется передача сообщения, но не выполняется его прием;
  - ❑ процесс-источник и процесс-получатель одновременно пытаются выполнить блокирующие передачу или прием сообщения.

# Двухточечный (point-to-point, p2p) обмен

```
#include <mpi.h>
int main (int argc, char **argv)
{
    int rank, size;
    int recv, send, left, right;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    left = (rank > 0)? rank - 1 : size-1;
    right = (rank + 1)%size ;
    MPI_Recv( &recv, 1, MPI_INT, left, 100, MPI_COMM_WORLD, &status );
    send = rank;
    MPI_Send( &send, 1, MPI_INT, right, 100, MPI_COMM_WORLD );
    MPI_Finalize( );
    return 0;
}
```

# Двухточечный (point-to-point, p2p) обмен

В MPI приняты следующие соглашения об именах подпрограмм двухточечного обмена:

**MPI\_[I] [R, S, B] Send**

здесь префикс [I] (Immediate) обозначает неблокирующий режим.

Один из префиксов [R, S, B] обозначает режим обмена: по готовности, синхронный и буферизованный.

Отсутствие префикса обозначает подпрограмму стандартного обмена.

Имеется 8 разновидностей операции передачи сообщений.

Для подпрограмм приема:

**MPI\_[I] Recv**

то есть всего 2 разновидности приема.

Подпрограмма MPI\_Irecv, например, выполняет передачу «по готовности» в неблокирующем режиме, MPI\_Brecv буферизованную передачу с блокировкой, а MPI\_Recv выполняет блокирующий прием сообщений.

Подпрограмма приема любого типа может принять сообщения от любой подпрограммы передачи.

# Стандартная блокирующая передача

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

**MPI\_Send(buf, count, datatype, dest, tag, comm,ierr)**

- ▶ buf – адрес первого элемента в буфере передачи;
- ▶ count – количество элементов в буфере передачи (допускается count = 0);
- ▶ datatype – тип MPI каждого пересылаемого элемента;
- ▶ dest – ранг процесса–получателя сообщения ( целое число от 0 до n – 1, где n – число процессов в области взаимодействия);
- ▶ tag – тег сообщения;
- ▶ comm – коммуникатор;
- ▶ ierr – код завершения.

# Стандартная блокирующая передача

- ▶ При стандартной блокирующей передаче после завершения вызова (после возврата из функции/процедуры передачи) можно использовать любые переменные, использовавшиеся в списке параметров. Такое использование не повлияет на корректность обмена.
- ▶ Дальнейшая «судьба» сообщения зависит от реализации MPI. Сообщение может быть сразу передано процессу-получателю или может быть скопировано в буфер передачи.
- ▶ Завершение вызова не гарантирует доставки сообщения по назначению.

# Стандартный блокирующий прием

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

```
MPI_Recv(buf, count, datatype, dest, tag, comm, status, ierr)
```

- ▶ buf – адрес первого элемента в буфере приёма;
- ▶ count – количество элементов в буфере приёма;
- ▶ datatype – тип MPI каждого пересылаемого элемента;
- ▶ source – ранг процесса-отправителя сообщения ( целое число от 0 до n – 1, где n – число процессов в области взаимодействия);
- ▶ tag – тег сообщения;
- ▶ comm – коммуникатор;
- ▶ status – статус обмена;
- ▶ ierr – код завершения.

# Стандартный блокирующий прием

- ▶ Значение параметра `count` может оказаться больше, чем количество элементов в принятом сообщении. В этом случае после выполнения приёма в буфере изменится значение только тех элементов, которые соответствуют элементам фактически принятого сообщения.
- ▶ Для функции `MPI_Recv` гарантируется, что после завершения вызова сообщение принято и размещено в буфере приема.

# Прием сообщения

Если один процесс последовательно посыпает два сообщения, соответствующие одному и тому же вызову MPI\_Recv, другому процессу, то первым будет принято сообщение, которое было отправлено раньше.

Если два сообщения были одновременно отправлены разными процессами, то порядок их получения принимающим процессом заранее не определён.

# Коды завершения

- ▶ MPI\_ERR\_COMM – неправильно указан коммуникатор.  
Часто возникает при использовании «пустого» коммуникатора;
- ▶ MPI\_ERR\_COUNT – неправильное значение аргумента count (количество пересылаемых значений);
- ▶ MPI\_ERR\_TYPE – неправильное значение аргумента, задающего тип данных;
- ▶ MPI\_ERR\_TAG – неправильно указан тег сообщения;
- ▶ MPI\_ERR\_RANK – неправильно указан ранг источника или адресата сообщения;
- ▶ MPI\_ERR\_ARG – неправильный аргумент, ошибочное задание которого не попадает ни в один класс ошибок;
- ▶ MPI\_ERR\_REQUEST – неправильный запрос на выполнение операции.

# Джокеры

- ▶ В качестве ранга источника сообщения и в качестве тега сообщения можно использовать «джокеры» :
  - MPI\_ANY\_SOURCE – любой источник;
  - MPI\_ANY\_TAG – любой тег.
- ▶ При использовании «джокеров» есть опасность приема сообщения, не предназначенного данному процессу

# Двухточечные обмены

Подпрограмма `MPI_Recv` может принимать сообщения, отправленные в любом режиме.

Прием может выполняться от произвольного процесса, а в операции передачи должен быть указан вполне определенный адрес.

Приемник может использовать «джокеры» для источника и для тега. Процесс может отправить сообщение и самому себе, но следует учитывать, что использование в этом случае блокирующих операций может привести к «тупику».

# Двухточечные обмены

Размер полученного сообщения (`count`) можно определить с помощью вызова подпрограммы

```
int MPI_Get_count(MPI_Status *status, MPI_Datatype  
datatype, int *count)
```

`MPI_Get_count(status, datatype, count, ierr)`

- ▶ `count` – количество элементов в буфере передачи;
- ▶ `datatype` – тип MPI каждого пересылаемого элемента;
- ▶ `status` – статус обмена;
- ▶ `ierr` – код завершения.

Аргумент `datatype` должен соответствовать типу данных, указанному в операции обмена

# Двухточечный обмен с буферизацией

- ▶ Передача сообщения в буферизованном режиме может быть начата независимо от того, зарегистрирован ли соответствующий прием. Источник копирует сообщение в буфер, а затем передает его в неблокирующем режиме, так же как в стандартном режиме.
- ▶ Эта операция локальна, поскольку ее выполнение не зависит от наличия соответствующего приема.
- ▶ Если объем буфера недостаточен, возникает ошибка. Выделение буфера и его размер контролируются программистом.

# Двухточечный обмен с буферизацией

- ▶ Размер буфера должен превосходить размер сообщения на величину MPI\_BSEND\_OVERHEAD. Это дополнительное пространство используется подпрограммой буферизованной передачи для своих целей.
- ▶ Если перед выполнением операции буферизованного обмена не выделен буфер, MPI ведет себя так, как если бы с процессом был связан буфер нулевого размера. Работа с таким буфером обычно завершается сбоем программы.
- ▶ Буферизованный обмен рекомендуется использовать в тех ситуациях, когда программисту требуется больший контроль над распределением памяти. Этот режим удобен и для отладки, поскольку причину переполнения буфера определить легче, чем причину тупика.

# Двухточечный обмен с буферизацией

- ▶ При выполнении буферизованного обмена программист должен заранее создать буфер достаточного размера:

```
int MPI_Buffer_attach(void *buf, size)
```

```
MPI_Buffer_attach(buf, size, ierr)
```

- ▶ В результате вызова создается буфер buf размером size байтов. В программах на языке Fortran роль буфера может играть массив. За один раз к процессу может быть подключен только один буфер.

# Двухточечный обмен с буферизацией

- ▶ Буферизованная передача завершается сразу, поскольку сообщение немедленно копируется в буфер для последующей передачи:

```
int MPI_Bsend(void *buf, int count,  
MPI_Datatype datatype, int dest, int tag,  
MPI_Comm comm)
```

```
MPI_Bsend(buf, count, datatype, dest, tag,  
comm, ierr)
```

# Двухточечный обмен с буферизацией

- ▶ После завершения работы с буфером его необходимо отключить:

```
int MPI_Buffer_detach(void *buf, int *size)
```

```
MPI_Buffer_detach(buf, size, ierr)
```

- ▶ Возвращается адрес (`buf`) и размер отключаемого буфера (`size`). Эта операция блокирует работу процесса до тех пор, пока все сообщения, находящиеся в буфере, не будут обработаны. Вызов данной подпрограммы можно использовать для форсированной передачи сообщений. После завершения вызова можно вновь использовать память, которую занимал буфер. В языке С данный вызов не освобождает автоматически память, отведенную для буфера.

# Двухточечный обмен с буферизацией

```
#include <mpi.h>
#include <stdio.h>
#define M 10
int main( int argc, char **argv )
{
    int n;
    int rank, size;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    if ( rank == 0 ) {
        int blen = M * (sizeof(int) + MPI_BSEND_OVERHEAD);
        int *buf = (int*) malloc(blen);
        MPI_Buffer_attach (buf, blen);
        for(int i = 0; i < M; i++) {
            n = i;
            MPI_Bsend (&n, 1, MPI_INT, 1, i, MPI_COMM_WORLD );
        }
        MPI_Buffer_detach(&buf, &blen);
        free(buf);
    }
    else if ( rank == 1 ) {
        for(int i = 0; i < M; i++) {
            MPI_Recv (&n, 1, MPI_INT, 0, i,
                      MPI_COMM_WORLD,&status );
        }
    }
    MPI_Finalize();
    return 0;
}
```

# Синхронный режим

- ▶ Завершение передачи происходит только после того, как прием сообщения инициализирован другим процессом.
- ▶ Посылающая сторона запрашивает у принимающей стороны подтверждение выдачи операции receive – «квитанцию».

```
int MPI_Ssend(void *buf, int count,  
MPI_Datatype datatype, int dest, int tag,  
MPI_Comm comm)
```

# Режим «по готовности»

Передача «по готовности» выполняется с помощью подпрограммы

```
int MPI_Rsend(void *buf, int count, MPI_Datatype  
datatype, int dest, int tag, MPI_Comm comm)
```

```
MPI_Rsend(buf, count, datatype, dest, tag, comm,  
ierr)
```

Передача «по готовности» должна начинаться, если уже зарегистрирован соответствующий прием. При несоблюдении этого условия результат выполнения операции не определен.

# Режим «ПО ГОТОВНОСТИ»

Завершается она сразу же. Если прием не зарегистрирован, результат выполнения операции не определен.

Завершение передачи не зависит от того, вызвана ли другим процессом подпрограмма приема данного сообщения или нет, оно означает только, что буфер передачи можно использовать вновь.

Сообщение просто выбрасывается в коммуникационную сеть в надежде, что адресат его получит. Эта надежда может и не сбыться.

# Режим «по готовности»

Обмен «по готовности» может увеличить производительность программы, поскольку здесь не используются этапы установки межпроцессных связей, а также буферизация.

Все это — операции, требующие времени. С другой стороны, обмен «по готовности» потенциально опасен, кроме того, он усложняет отладку, поэтому его рекомендуется использовать только в том случае, когда правильная работа программы гарантируется ее логической структурой, а выигрыша в быстродействии надо добиться любой ценой.

# Совместные прием и передача

- ▶ Подпрограмма MPI\_Sendrecv выполняет прием и передачу данных с блокировкой:

```
int MPI_Sendrecv(void *sendbuf, int sendcount,  
MPI_Datatype sendtype, int dest, int sendtag, void  
*recvbuf, int recvcount, MPI_Datatype recvtype,  
int source, int recvtag, MPI_Comm comm, MPI_Status  
*status)
```

- ▶ Подпрограмма MPI\_Sendrecv\_replace выполняет прием и передачу данных, используя общий буфер для передачи и приёма:

```
int MPI_Sendrecv_replace(void *buf, int count,  
MPI_Datatype datatype, int dest, int sendtag, int  
source, int recvtag, MPI_Comm comm, MPI_Status  
*status)
```

# Неблокирующие обмены

- ▶ Вызов подпрограммы неблокирующей передачи инициирует, но не завершает ее. Завершиться выполнение подпрограммы может еще до того, как сообщение будет скопировано в буфер передачи.
- ▶ Применение неблокирующих операций улучшает производительность программы, поскольку в этом случае допускается перекрытие (то есть одновременное выполнение) вычислений и обменов. Передача данных из буфера или их считывание может происходить одновременно с выполнением процессом другой работы.

# Неблокирующие обмены

- ▶ Для завершения неблокирующего обмена требуется вызов дополнительной процедуры, которая проверяет, скопированы ли данные в буфер передачи.
- ▶ **ВНИМАНИЕ!**  
При неблокирующем обмене возвращение из подпрограммы обмена происходит сразу, но запись в буфер или считывание из него после этого производить **нельзя** – сообщение может быть еще не отправлено или не получено и работа с буфером может «испортить» его содержимое.

# Неблокирующие обмены

Неблокирующий обмен выполняется в два этапа:

- 1. инициализация обмена;**
- 2. проверка завершения обмена.**

Разделение этих шагов делает необходимым **маркировку** каждой операции обмена, которая позволяет целенаправленно выполнять проверки завершения соответствующих операций.

Для маркировки в неблокирующих операциях используются *идентификаторы операций обмена*

# Неблокирующие обмены

- ▶ Инициализация неблокирующей стандартной передачи выполняется подпрограммами MPI\_I[S, B, R]send.  
Стандартная неблокирующая передача выполняется подпрограммой:

```
int MPI_Isend(void *buf, int count, MPI_Datatype  
datatype, int dest, int tag, MPI_Comm comm,  
MPI_Request *request)
```

```
MPI_Isend(buf, count, datatype, dest, tag, comm,  
request, ierr)
```

- ▶ Входные параметры этой подпрограммы аналогичны аргументам подпрограммы MPI\_Send.
- ▶ Выходной параметр request – идентификатор операции.

# Неблокирующие обмены

- ▶ Инициализация неблокирующего приема выполняется при вызове подпрограммы:

```
int MPI_Irecv(void *buf, int count,  
MPI_Datatype datatype, int source, int tag,  
MPI_Comm comm, MPI_Request *request)
```

```
MPI_Irecv(buf, count, datatype, source, tag,  
comm, request, ierr)
```

- ▶ Назначение аргументов здесь такое же, как и в ранее рассмотренных подпрограммах, за исключением того, что указывается ранг не адресата, а источника сообщения (`source`).

# Неблокирующие обмены

- ▶ Вызовы подпрограмм неблокирующего обмена формируют *запрос* на выполнение операции обмена и связывают его с идентификатором операции *request*.
- ▶ Запрос идентифицирует свойства операции обмена:
  - режим;
  - характеристики буфера обмена;
  - контекст;
  - тег и ранг.
- ▶ Запрос содержит информацию о состоянии ожидающих обработки операций обмена и может быть использован для получения информации о состоянии обмена или для ожидания его завершения.

# Проверка выполнения обмена

- ▶ Проверка фактического выполнения передачи или приема в неблокирующем режиме осуществляется с помощью вызова *подпрограмм ожидания*, блокирующих работу процесса до завершения операции или неблокирующих подпрограмм проверки, возвращающих логическое значение «истина», если операция выполнена

# Проверка выполнения обмена

- ▶ В том случае, когда одновременно несколько процессов обмениваются сообщениями, можно использовать проверки, которые применяются одновременно к нескольким обменам.
- ▶ Есть три типа таких проверок:
  1. проверка завершения всех обменов;
  2. проверка завершения любого обмена из нескольких;
  3. проверка завершения заданного обмена из нескольких.
- ▶ Каждая из этих проверок имеет две разновидности:
  1. «ожидание»;
  2. «проверка».

# Блокирующие операции проверки

- ▶ Подпрограмма MPI\_Wait блокирует работу процесса до завершения приема или передачи сообщения:

```
int MPI_Wait(MPI_Request *request, MPI_Status  
*status)
```

```
MPI_Wait(request, status, ierr)
```

- ▶ Входной параметр request — идентификатор операции обмена, выходной — статус (status).

# Блокирующие операции проверки

- ▶ Успешное выполнение подпрограммы MPI\_Wait после вызова MPI\_Ibsend подразумевает, что буфер передачи можно использовать вновь, то есть пересылаемые данные отправлены или скопированы в буфер, выделенный при вызове подпрограммы MPI\_Buffer\_attach.
- ▶ В этот момент уже нельзя отменить передачу. Если не будет зарегистрирован соответствующий прием, буфер нельзя будет освободить. В этом случае можно применить подпрограмму MPI\_Cancel, которая освобождает память, выделенную подсистеме коммуникаций.

# Проверка завершения всех обменов

- ▶ Проверка завершения всех обменов выполняется подпрограммой:

```
int MPI_Waitall(int count, MPI_Request  
requests[], MPI_Status statuses[])
```

```
MPI_Waitall(count, requests, statuses, ierr)
```

- ▶ При вызове этой подпрограммы выполнение процесса блокируется до тех пор, пока все операции обмена, связанные с активными запросами в массиве `requests`, не будут выполнены. Возвращается статус этих операций. Статус обменов содержится в массиве `statuses`. `count` – количество запросов на обмен (размер массивов `requests` и `statuses`).

# Проверка завершения всех обменов

- ▶ В результате выполнения подпрограммы MPI\_Waitall запросы, сформированные неблокирующими операциями обмена, аннулируются, а соответствующим элементам массива присваивается значение MPI\_REQUEST\_NULL.
- ▶ В случае неуспешного выполнения одной или более операций обмена подпрограмма MPI\_Waitall возвращает код ошибки MPI\_ERR\_IN\_STATUS и присваивает полю ошибки статуса значение кода ошибки соответствующей операции.
- ▶ Если операция выполнена успешно, полю присваивается значение MPI\_SUCCESS, а если не выполнена, но и не было ошибки – значение MPI\_ERR\_PENDING. Это соответствует наличию запросов на выполнение операции обмена, ожидающих обработки.

# Алгоритм Якоби. Последовательная версия

```
/* Jacobi program */  
#include <stdio.h>  
#define L 1000  
#define ITMAX 100  
int i,j,it;  
double A[L][L];  
double B[L][L];  
int main(int an, char **as)  
{  
    printf("JAC STARTED\n");  
    for(i=0;i<=L-1;i++)  
        for(j=0;j<=L-1;j++)  
        {  
            A[i][j]=0.;  
            B[i][j]=1.+i+j;  
        }  
}
```

# Алгоритм Якоби. Последовательная версия

```
***** iteration loop *****/
for(it=1; it<ITMAX;it++)
{
    for(i=1;i<=L-2;i++)
        for(j=1;j<=L-2;j++)
            A[i][j] = B[i][j];
    for(i=1;i<=L-2;i++)
        for(j=1;j<=L-2;j++)
            B[i][j] = (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4.;
}
return 0;
}
```

# Алгоритм Якоби. MPI-версия

$A_{00}$	$A_{01}$	$A_{02}$	$A_{03}$	$A_{04}$	$A_{05}$	$A_{06}$	$A_{07}$	$A_{08}$
$A_{10}$	$A_{11}$	$A_{12}$	$A_{13}$	$A_{14}$	$A_{15}$	$A_{16}$	$A_{17}$	$A_{18}$
$A_{20}$	$A_{21}$	$A_{22}$	$A_{23}$	$A_{24}$	$A_{25}$	$A_{26}$	$A_{27}$	$A_{28}$
$A_{30}$	$A_{31}$	$A_{32}$	$A_{33}$	$A_{34}$	$A_{35}$	$A_{36}$	$A_{37}$	$A_{38}$



Shadow edges

$A_{20}$	$A_{21}$	$A_{22}$	$A_{23}$	$A_{24}$	$A_{25}$	$A_{26}$	$A_{27}$	$A_{28}$
$A_{30}$	$A_{31}$	$A_{32}$	$A_{33}$	$A_{34}$	$A_{35}$	$A_{36}$	$A_{37}$	$A_{38}$



Imported elements

$A_{40}$	$A_{41}$	$A_{42}$	$A_{43}$	$A_{44}$	$A_{45}$	$A_{46}$	$A_{47}$	$A_{48}$
$A_{50}$	$A_{51}$	$A_{52}$	$A_{53}$	$A_{54}$	$A_{55}$	$A_{56}$	$A_{57}$	$A_{58}$

$A_{60}$	$A_{61}$	$A_{62}$	$A_{63}$	$A_{64}$	$A_{65}$	$A_{66}$	$A_{67}$	$A_{68}$
$A_{50}$	$A_{51}$	$A_{52}$	$A_{53}$	$A_{54}$	$A_{55}$	$A_{56}$	$A_{57}$	$A_{58}$

$A_{60}$	$A_{61}$	$A_{62}$	$A_{63}$	$A_{64}$	$A_{65}$	$A_{66}$	$A_{67}$	$A_{68}$
$A_{70}$	$A_{71}$	$A_{72}$	$A_{73}$	$A_{74}$	$A_{75}$	$A_{76}$	$A_{77}$	$A_{78}$

$A_{80}$	$A_{81}$	$A_{82}$	$A_{83}$	$A_{84}$	$A_{85}$	$A_{86}$	$A_{87}$	$A_{88}$
$A_{50}$	$A_{51}$	$A_{52}$	$A_{53}$	$A_{54}$	$A_{55}$	$A_{56}$	$A_{57}$	$A_{58}$

# Алгоритм Якоби. MPI-версия

```
/* Jacobi-1d program */
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include "mpi.h"
#define m_printf if (myrank==0)printf
#define L 1000
#define ITMAX 100

int i,j,it,k;
int II,shift;
double (* A)[L];
double (* B)[L];
```

# Алгоритм Якоби. MPI-версия

```
int main(int argc, char **argv)
{
    MPI_Request req[4];
    int myrank, ranksize;
    int startrow, lastrow, nrow;
    MPI_Status status[4];
    double t1, t2, time;
    MPI_Init (&argc, &argv); /* initialize MPI system */
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/*my place in MPI system*/
    MPI_Comm_size (MPI_COMM_WORLD, &ranksize); /* size of MPI system */
    MPI_Barrier(MPI_COMM_WORLD);
    /* rows of matrix I have to process */
    startrow = (myrank *L) / ranksize;
    lastrow = (((myrank + 1) * L) / ranksize)-1;
    nrow = lastrow - startrow + 1;
    m_printf("JAC1 STARTED\n");
```

# Алгоритм Якоби. MPI-версия

```
/* dynamically allocate data structures */
A = malloc ((nrow+2) * L * sizeof(double));
B = malloc ((nrow) * L * sizeof(double));
for(i=1; i<=nrow; i++)
    for(j=0; j<=L-1; j++)
    {
        A[i][j]=0.;
        B[i-1][j]=1.+startrow+i-1+j;
    }
```

# Алгоритм Якоби. MPI-версия

```
***** iteration loop *****/
t1=MPI_Wtime();
for(it=1; it<=ITMAX; it++)
{
    for(i=1; i<=nrow; i++)
    {
        if (((i==1)&&(myrank==0))||((i==nrow)&&(myrank==ranksize-1)))
            continue;
        for(j=1; j<=L-2; j++)
        {
            A[i][j] = B[i-1][j];
        }
    }
}
```

# Алгоритм Якоби. MPI-версия

```
if(myrank!=0)
    MPI_Irecv(&A[0][0],L,MPI_DOUBLE, myrank-1, 1215,
              MPI_COMM_WORLD, &req[0]);
if(myrank!=ranksize-1)
    MPI_Isend(&A[nrow][0],L,MPI_DOUBLE, myrank+1, 1215,
              MPI_COMM_WORLD,&req[2]);
if(myrank!=ranksize-1)
    MPI_Irecv(&A[nrow+1][0],L,MPI_DOUBLE, myrank+1, 1216,
              MPI_COMM_WORLD, &req[3]);
if(myrank!=0)
    MPI_Isend(&A[1][0],L,MPI_DOUBLE, myrank-1, 1216,
              MPI_COMM_WORLD,&req[1]);
ll=4; shift=0;
if (myrank==0) {ll=2;shift=2;}
if (myrank==ranksize-1) {ll=2;}
MPI_Waitall(ll,&req[shift],&status[0]);
```

# Алгоритм Якоби. MPI-версия

```
if(myrank!=0)
    MPI_Recv(&A[0][0],L,MPI_DOUBLE, myrank-1, 1215,
             MPI_COMM_WORLD, &status[0]);
if(myrank!=ranksize-1)
    MPI_Send(&A[nrow][0],L,MPI_DOUBLE, myrank+1, 1215,
             MPI_COMM_WORLD);
if(myrank!=ranksize-1)
    MPI_Recv(&A[nrow+1][0],L,MPI_DOUBLE, myrank+1, 1216,
             MPI_COMM_WORLD, &status[0]);
if(myrank!=0)
    MPI_Send(&A[1][0],L,MPI_DOUBLE, myrank-1, 1216,
             MPI_COMM_WORLD);
```

# Алгоритм Якоби. MPI-версия

```
for(i=1; i<=nrow; i++)
{
    if (((i==1)&&(myrank==0))||((i==nrow)&&(myrank==ranksize-1))) continue;
    for(j=1; j<=L-2; j++)
        B[i-1][j] = (A[i-1][j]+A[i+1][j]+
                      A[i][j-1]+A[i][j+1])/4.;

    }
/*DO it*/
printf("%d: Time of task=%lf\n",myrank,MPI_Wtime()-t1);
MPI_Finalize ();
return 0;
}
```

# Проверка завершения любого числа обменов

- ▶ Проверка завершения любого числа обменов выполняется подпрограммой:

```
int MPI_Waitany(int count, MPI_Request requests[],  
int *index, MPI_Status *status)
```

- ▶ Выполнение процесса блокируется до тех пор, пока, по крайней мере, один обмен из массива запросов (`requests`) не будет завершен.
- ▶ Входные параметры:
  - `requests` – запрос;
  - `count` – количество элементов в массиве `requests`.
- ▶ Выходные параметры:
  - `index` – индекс запроса (в языке С это целое число от 0 до `count - 1`) в массиве `requests`;
  - `status` – статус.

# Неблокирующие процедуры проверки

- ▶ Подпрограмма MPI\_Test выполняет неблокирующую проверку завершения приема или передачи сообщения:

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

- ▶ Входной параметр: идентификатор операции обмена request.
- ▶ Выходные параметры:
  - ❑ flag — «истина», если операция, заданная идентификатором request, выполнена;
  - ❑ status — статус выполненной операции.

# Неблокирующая проверка завершения всех обменов

- ▶ Подпрограмма MPI\_Testall выполняет неблокирующую проверку завершения приема или передачи всех сообщений:

```
int MPI_Testall(int count, MPI_Request requests[],  
int *flag, MPI_Status statuses[])
MPI_Testall(count, requests, flag, statuses, ierr)
```

- ▶ При вызове возвращается значение флага (`flag`) «истина», если все обмены, связанные с активными запросами в массиве `requests`, выполнены. Если завершены не все обмены, флагу присваивается значение «ложь», а массив `statuses` не определен.
- ▶ Параметр `count` – количество запросов.
- ▶ Каждому статусу, соответствующему активному запросу, присваивается значение статуса соответствующего обмена.

# Неблокирующая проверка любого числа обменов

- ▶ Подпрограмма MPI\_Testany выполняет неблокирующую проверку завершения приема или передачи сообщения:

```
int MPI_Testany(int count, MPI_Request  
requests[], int *index, int *flag, MPI_Status  
*status)
```

- ▶ Смысл и назначение параметров этой подпрограммы те же, что и для подпрограммы MPI\_Waitany.  
Дополнительный аргумент flag, принимает значение «истина», если одна из операций завершена.
- ▶ Блокирующая подпрограмма MPI\_Waitany и неблокирующая MPI\_Testany взаимозаменяемы, как и другие аналогичные пары.

# Другие операции проверки

- ▶ Подпрограммы MPI\_Waitsome И MPI\_Testsome действуют аналогично подпрограммам MPI\_Waitany И MPI\_Testany, кроме случая, когда завершается более одного обмена.
- ▶ В подпрограммах MPI\_Waitany И MPI\_Testany обмен из числа завершенных выбирается произвольно, именно для него и возвращается статус, а для MPI\_Waitsome И MPI\_Testsome статус возвращается для всех завершенных обменов.
- ▶ Эти подпрограммы можно использовать для определения, сколько обменов завершено.

# Другие операции проверки

- ▶ Блокирующая проверка выполнения обменов:

```
int MPI_Waitsome(int incount, MPI_Request requests[], int *outcount, int indices[], MPI_Status statuses[])
```

- ▶ Здесь incount – количество запросов. В outcount возвращается количество выполненных запросов из массива requests, а в первых outcount элементах массива indices возвращаются индексы этих операций. В первых outcount элементах массива statuses возвращается статус завершенных операций. Если выполненный запрос был сформирован неблокирующей операцией обмена, он аннулируется. Если в списке нет активных запросов, выполнение подпрограммы завершается сразу, а параметру outcount присваивается значение MPI\_UNDEFINED.

# Другие операции проверки

- ▶ Неблокирующая проверка выполнения обменов:

```
int MPI_Testsome(int incount, MPI_Request  
requests[], int *outcount, int indices[],  
MPI_Status statuses[])
```

- ▶ Параметры такие же, как и у подпрограммы MPI\_Waitsome. Эффективность подпрограммы MPI\_Testsome выше, чем у MPI\_Testany, поскольку первая возвращает информацию обо всех операциях, а для второй требуется новый вызов для каждой выполненной операции.

# Проверка статуса операции приема сообщения

- ▶ Блокирующая проверка:

```
int MPI_Probe(int source, int tag, MPI_Comm comm,  
              MPI_Status* status)
```

- ▶ Неблокирующая проверка:

```
int MPI_Iprobe(int source, int tag, MPI_Comm comm,  
               int *flag, MPI_Status *status)
```

- ▶ Входные параметры этой подпрограммы те же, что и у подпрограммы MPI\_Probe.
- ▶ Выходные параметры:
  - ❑ flag – флаг;
  - ❑ status – статус.
- ▶ Если сообщение уже поступило и может быть принято, возвращается значение флага «истина».

# Проверка статуса операции приема сообщения (пример)

```
if (rank == 0) {  
    MPI_Send(buf, size, MPI_INT, 1, 0, MPI_COMM_WORLD); // Send to process 1  
    printf("0 sent %d numbers to 1\n", size);  
} else if (rank == 1) {  
    MPI_Status status;  
    MPI_Probe(0, 0, MPI_COMM_WORLD, &status); // Probe for an incoming msg.  
    // When probe returns, the status object has the size and other  
    // attributes of the incoming message. Get the size of the message.  
    MPI_Get_count(&status, MPI_INT, &size);  
    // Allocate a buffer just big enough to hold the incoming numbers  
    int* bufnumber = (int*)malloc(sizeof(int) * size);  
    // Now receive the message with the allocated buffer  
    MPI_Recv(bufnumber, size, MPI_INT, 0, 0,  
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);  
    printf("1 dynamically received %d numbers from 0.\n", size);  
    free(bufnumber);  
}
```

# Коллективные операции

- ▶ Передача сообщений между группой процессов
- ▶ Вызываются ВСЕМИ процессами в коммуникаторе
- ▶ Примеры:
  - Broadcast, scatter, gather (рассылка данных)
  - Global sum, global maximum, и.т.д.  
(редукционные операции)
  - Барьерная синхронизация

# Характеристики коллективных передач

- ▶ Коллективные операции не являются помехой операциям типа точка–точка и наоборот
- ▶ Все процессы коммуникатора должны вызывать коллективную операцию
- ▶ Синхронизация не гарантируется (за исключением барьера)
- ▶ Нет тэгов
- ▶ Принимающий буфер должен точно соответствовать размеру отсылаемого буфера

# Особенности коллективных передач

```
switch(rank) {  
    case 0:  
        MPI_Bcast(buf1, count, type, 0, comm);  
        MPI_Bcast(buf2, count, type, 1, comm);  
        break;  
    case 1:  
        MPI_Bcast(buf2, count, type, 1, comm);  
        MPI_Bcast(buf1, count, type, 0, comm);  
        break;  
}
```

# Особенности коллективных передач

```
switch(rank) {  
    case 0:  
        MPI_Bcast(buf1, count, type, 0, comm0);  
        MPI_Bcast(buf2, count, type, 2, comm2);  
        break;  
    case 1:  
        MPI_Bcast(buf1, count, type, 1, comm1);  
        MPI_Bcast(buf2, count, type, 0, comm0);  
        break;  
    case 2:  
        MPI_Bcast(buf1, count, type, 2, comm2);  
        MPI_Bcast(buf2, count, type, 1, comm1);  
        break;  
}  
/* comm0 is {0,1}, comm1 is {1, 2} comm2 is {2,0} */
```

# Особенности коллективных передач

```
switch(rank) {  
    case 0:  
        MPI_Bcast(buf1, count, type, 0, comm);  
        MPI_Send(buf2, count, type, 1, tag, comm);  
        break;  
    case 1:  
        MPI_Recv(buf2, count, type, 0, tag, comm, status);  
        MPI_Bcast(buf1, count, type, 0, comm);  
        break;  
}
```

# Обзор коллективных операций

- ▶ Синхронизация всех процессов с помощью барьеров (MPI\_Barrier).
- ▶ Коллективные коммуникационные операции, в число которых входят:
  - рассылка информации от одного процесса всем остальным членам некоторой области связи (MPI\_Bcast);
  - сборка распределенного по процессам массива в один массив с сохранением его в адресном пространстве выделенного (root) процесса (MPI\_Gather, MPI\_Gatherv);
  - сборка распределенного массива в один массив с рассылкой его всем процессам некоторой области связи (MPI\_Allgather, MPI\_Allgatherv);
  - разбиение массива и рассылка его фрагментов (scatter) всем процессам области связи (MPI\_Scatter, MPI\_Scatterv);
  - совмещенная операция Scatter/Gather (All-to-All), каждый процесс делит данные из своего буфера передачи и разбрасывает фрагменты всем остальным процессам, одновременно собирая фрагменты, посланные другими процессами в свой буфер приема (MPI\_Alltoall, MPI\_Alltoallv).

# Обзор коллективных операций

- ▶ Глобальные вычислительные операции (sum, min, max и др.) над данными, расположенными в адресных пространствах различных процессов:
  - с сохранением результата в адресном пространстве одного процесса (MPI\_Reduce);
  - с рассылкой результата всем процессам (MPI\_Allreduce);
  - совмещенная операция Reduce/Scatter (MPI\_Reduce\_scatter);
  - префиксная редукция (MPI\_Scan).
- ▶ Все коммуникационные подпрограммы, за исключением MPI\_Bcast, представлены в двух вариантах:
  - простой вариант, когда все части передаваемого сообщения имеют одинаковую длину;
  - "векторный" вариант, который снимает ограничения, присущие простому варианту, как в части длин блоков, так и в части размещения данных в адресном пространстве процессов.  
Векторные варианты отличаются дополнительным символом "v" в конце имени функции.

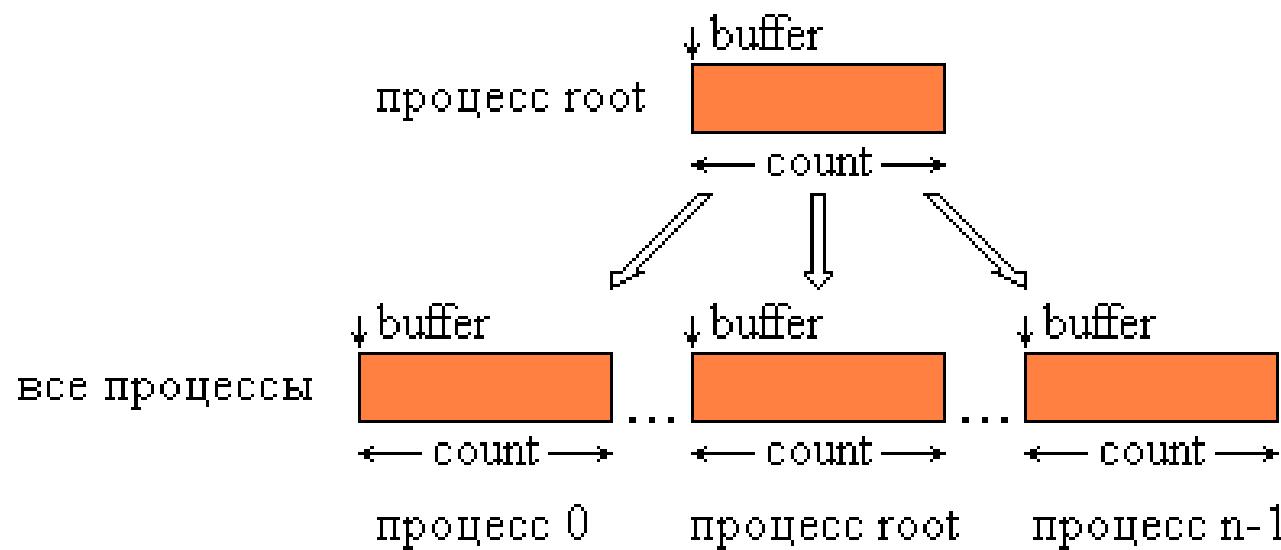
# Широковещательная рассылка

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

INOUT buffer - адрес начала расположения в памяти рассылаемых данных;  
IN count - число посылаемых элементов;  
IN datatype - тип посылаемых элементов;  
IN root - номер процесса-отправителя;  
IN comm - коммуникатор.

Процесс с номером root рассыпает сообщение из своего буфера передачи всем процессам области связи коммуникатора comm.

# Широковещательная рассылка



# Сбор блоков данных от всех процессов группы

Сборка блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером root:

```
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype,  
               void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

IN sendbuf - адрес начала размещения посылаемых данных;

IN sendcount - число посылаемых элементов;

IN sendtype - тип посылаемых элементов;

OUT recvbuf - адрес начала буфера приема (используется только в процессе-получателе root);

IN recvcount - число элементов, получаемых от каждого процесса (используется только в процессе-получателе root);

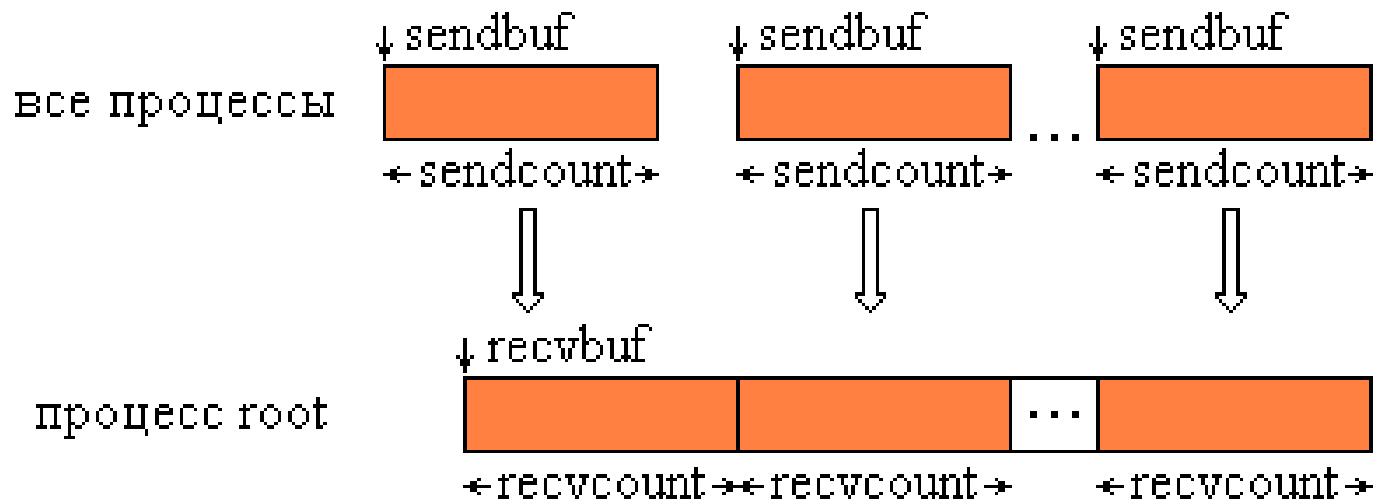
IN recvtype - тип получаемых элементов;

IN root - номер процесса-получателя; IN comm - коммуникатор.

# Сбор блоков данных от всех процессов группы

- ▶ Функция **MPI\_Gather** производит сборку блоков данных, посылаемых всеми процессами группы, в один массив процесса с номером root.
- ▶ Длина блоков предполагается одинаковой.
- ▶ Объединение происходит в порядке увеличения номеров процессов-отправителей. То есть данные, посланные процессом  $i$  из своего буфера sendbuf, помещаются в  $i$ -ю порцию буфера recvbuf процесса root.
- ▶ Длина массива, в который собираются данные, должна быть достаточной для их размещения.

# Сбор блоков данных от всех процессов группы



# Сбор блоков данных от всех процессов группы

Функция **MPI\_Gatherv** позволяет собирать блоки с разным числом элементов:

```
int MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,  
void* rbuf, int *recvcounts, int *displs, MPI_Datatype recvtype,  
int root, MPI_Comm comm)
```

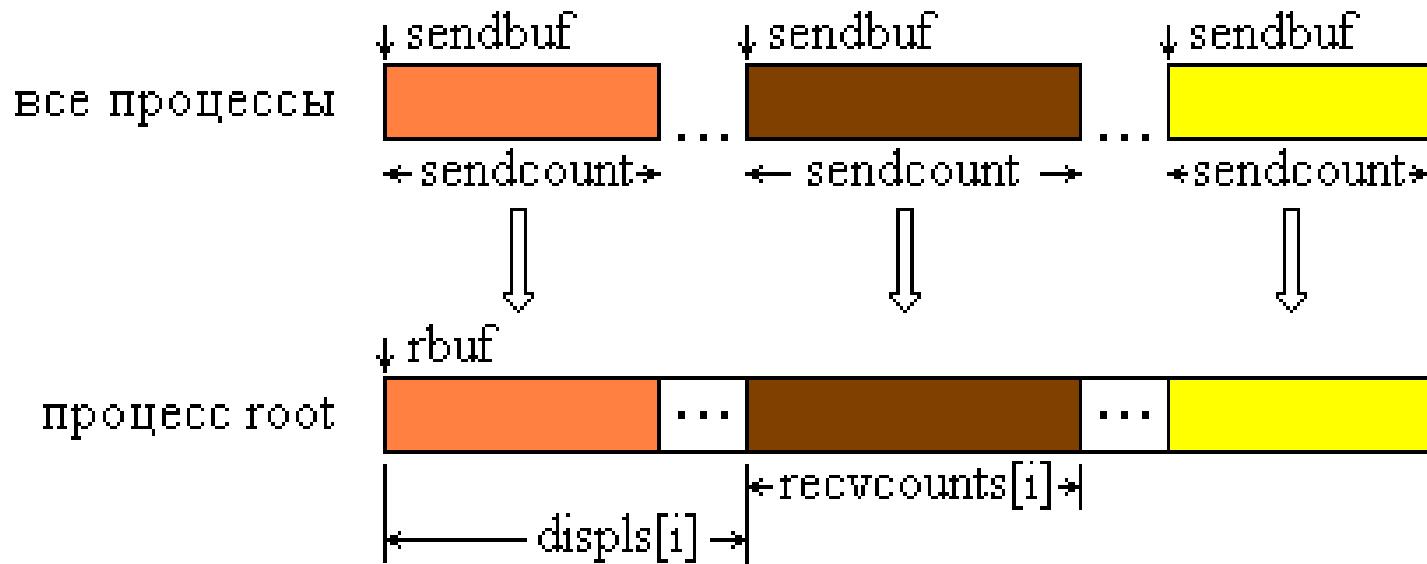
- IN sendbuf - адрес начала буфера передачи;
- IN sendcount - число посылаемых элементов;
- IN sendtype - тип посылаемых элементов;
- OUT rbuf - адрес начала буфера приема;
- IN recvcounts - целочисленный массив (размер равен числу процессов в группе), i-й элемент которого определяет число элементов, которое должно быть получено от процесса i;
- IN displs - целочисленный массив (размер равен числу процессов в группе), i-ое значение определяет смещение i-го блока данных относительно начала rbuf;
- IN recvtype - тип получаемых элементов;
- IN root - номер процесса-получателя;
- IN comm - коммуникатор.

# Сбор блоков данных от всех процессов группы

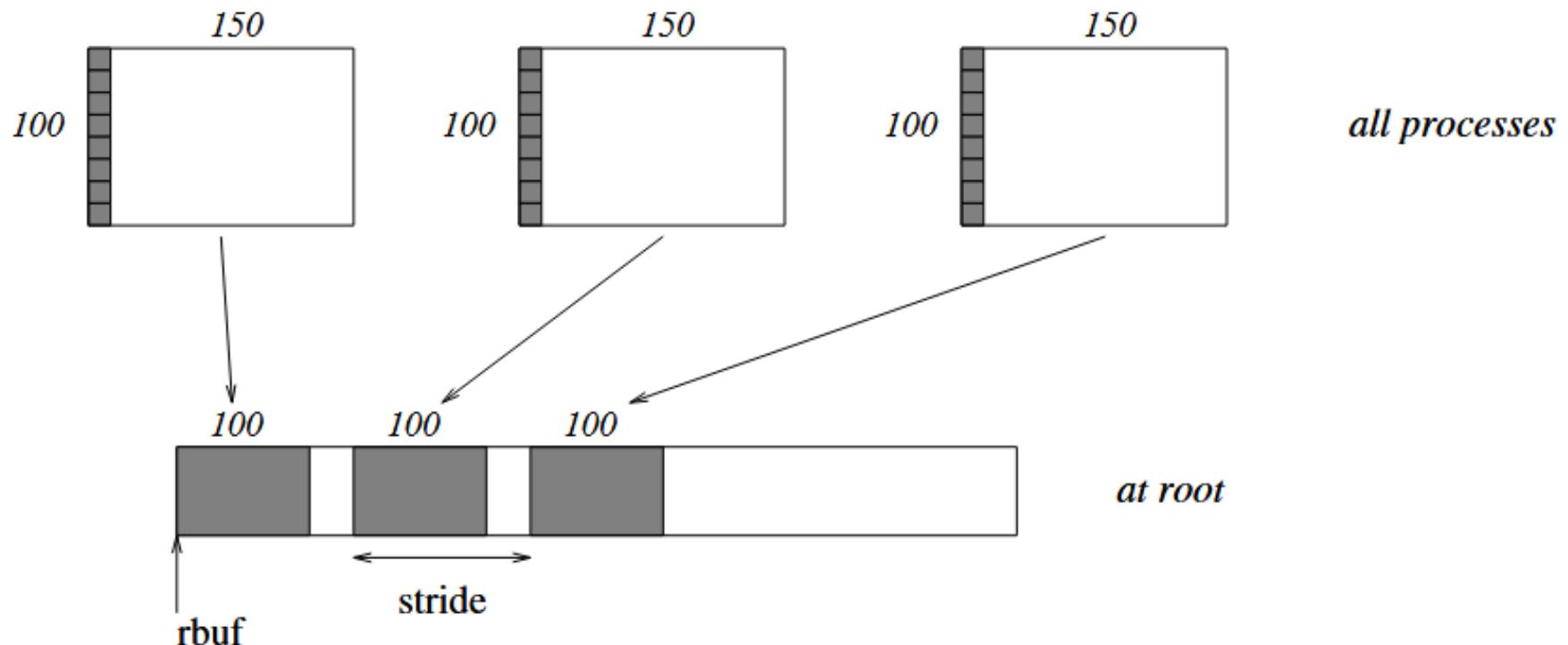
Функция **`MPI_Gatherv`** позволяет собирать блоки с разным числом элементов от каждого процесса, так как количество элементов, принимаемых от каждого процесса, задается индивидуально с помощью массива `recvcounts`. Эта функция обеспечивает также большую гибкость при размещении данных в процессе-получателе, благодаря введению в качестве параметра массива смещений `displs`.

Сообщения помещаются в буфер приема процесса `root` в соответствии с номерами посылающих процессов, а именно, данные, посланные процессом `i`, размещаются в адресном пространстве процесса `root`, начиная с адреса `rbuf + displs[i]`

# Сбор блоков данных от всех процессов группы



# Сбор блоков данных от всех процессов группы



# Сбор блоков данных от всех процессов группы

```
MPI_Comm_size(comm, &gsize);
rbuf = (int *)malloc(gsize*stride*sizeof(int));
displs = (int *)malloc(gsize*sizeof(int));
rcounts = (int *)malloc(gsize*sizeof(int));
for (i=0; i<gsize; ++i) {
    displs[i] = i*stride;
    rcounts[i] = 100;
}
/* Create datatype for 1 column of array */
MPI_Type_vector(100, 1, 150, MPI_INT, &stype);
MPI_Type_commit(&stype);

MPI_Gatherv(sendarray, 1, stype, rbuf, rcounts, displs, MPI_INT, root, comm);
```

# Сбор блоков данных от всех процессов группы

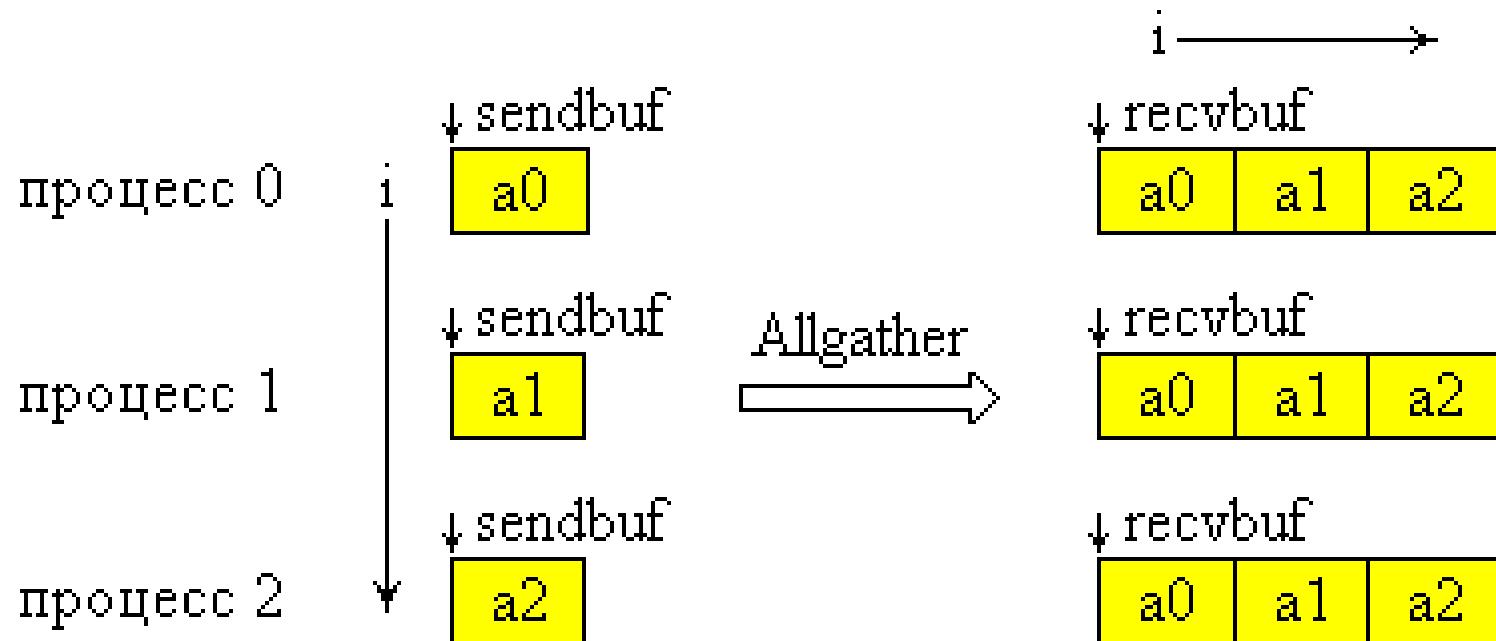
Функция **MPI\_Allgather** выполняется так же, как **MPI\_Gather**, но получателями являются все процессы группы. Данные, посланные процессом  $i$  из своего буфера `sendbuf`, помещаются в  $i$ -ю порцию буфера `recvbuf` каждого процесса. После завершения операции содержимое буферов приема `recvbuf` у всех процессов одинаково.

C:

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype,  
void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

IN	sendbuf	-	адрес начала буфера посылки;
IN	sendcount	-	число посылаемых элементов;
IN	sendtype	-	тип посылаемых элементов;
OUT	recvbuf	-	адрес начала буфера приема;
IN	recvcount	-	число элементов, получаемых от каждого процесса;
IN	recvtype	-	тип получаемых элементов;
IN	comm	-	коммуникатор.

# Сбор блоков данных от всех процессов группы



# Сбор блоков данных от всех процессов группы

Функция **MPI\_Allgatherv** является аналогом функции **MPI\_Gatherv**, но сборка выполняется всеми процессами группы. Поэтому в списке параметров отсутствует параметр root.

```
int MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype,  
void* rbuf, int *recvcounts, int *displs,  
MPI_Datatype recvtype, MPI_Comm comm)
```

IN sendbuf	-	адрес начала буфера передачи;
IN sendcount	-	число посылаемых элементов;
IN sendtype	-	тип посылаемых элементов;
OUT rbuf	-	адрес начала буфера приема;
IN recvcounts	-	целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, которое должно быть получено от каждого процесса;
IN displs	-	целочисленный массив (размер равен числу процессов в группе), i-ое значение определяет смещение относительно начала rbuf i-го блока данных;
IN recvtype	-	тип получаемых элементов; IN comm - коммуникатор.

# Рассылка блоков данных по всем процессам группы

Функция **MPI\_Scatter** разбивает сообщение из буфера посылки процесса root на равные части размером sendcount и посыпает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе).

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,  
void* recvbuf, int recvcount, MPI_Datatype recvtype,  
int root, MPI_Comm comm)
```

- |              |  |
|--------------|--|
| IN sendbuf   | - адрес начала размещения блоков распределяемых данных<br>(используется только в процессе-отправителе root); |
| IN sendcount | - число элементов, посылаемых каждому процессу;  |
| IN sendtype  | - тип посылаемых элементов;  |
| OUT recvbuf  | - адрес начала буфера приема;  |
| IN recvcount | - число получаемых элементов;  |
| IN recvtype  | - тип получаемых элементов;  |
| IN root      | - номер процесса-отправителя;  |
| IN comm      | - коммуникатор.  |

# Рассылка блоков данных по всем процессам группы

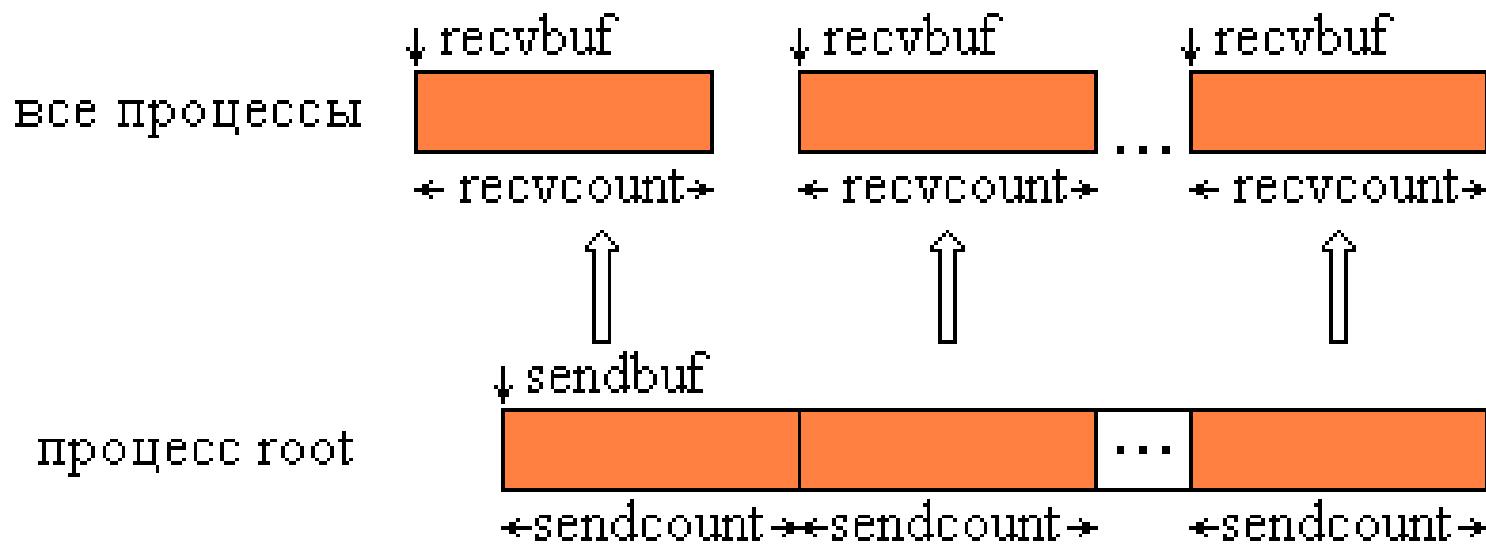
Функция **MPI\_Scatter** разбивает сообщение из буфера посылки процесса root на равные части размером sendcount и посыпает i-ю часть в буфер приема процесса с номером i (в том числе и самому себе).

Процесс root использует оба буфера (посылки и приема), поэтому в вызываемой им подпрограмме все параметры являются существенными. Остальные процессы группы с коммуникатором comm являются только получателями, поэтому для них параметры, специфицирующие буфер посылки, не существенны.

Число посыпаемых элементов sendcount должно равняться числу принимаемых recvcount.

Следует обратить внимание, что значение sendcount в вызове из процесса root - это число посыпаемых каждому процессу элементов, а не общее их количество. Операция Scatter является обратной по отношению к Gather.

# Рассылка блоков данных по всем процессам группы



# Рассылка блоков данных по всем процессам группы

Функция **MPI\_Scaterv** является векторным вариантом функции **MPI\_Scatter**, позволяющим посыпать каждому процессу различное количество элементов. Начало расположения элементов блока, посылаемого  $i$ -му процессу, задается в массиве смещений **displs**, а число посылаемых элементов - в массиве **sendcounts**.

```
int MPI_Scaterv(void* sendbuf, int *sendcounts, int *displs,  
                 MPI_Datatype sendtype, void* recvbuf, int recvcount,  
                 MPI_Datatype recvtype, int root, MPI_Comm comm)
```

IN **sendbuf** - адрес начала буфера посылки (используется только в процессе-отправителе **root**);

IN **sendcounts** - целочисленный массив (размер равен числу процессов в группе), содержащий число элементов, посылаемых каждому процессу;

IN **displs** - целочисленный массив (размер равен числу процессов в группе),  $i$ -ое значение определяет смещение относительно начала **sendbuf** для данных, посылаемых процессу  $i$ ;

IN **sendtype** - тип посылаемых элементов;

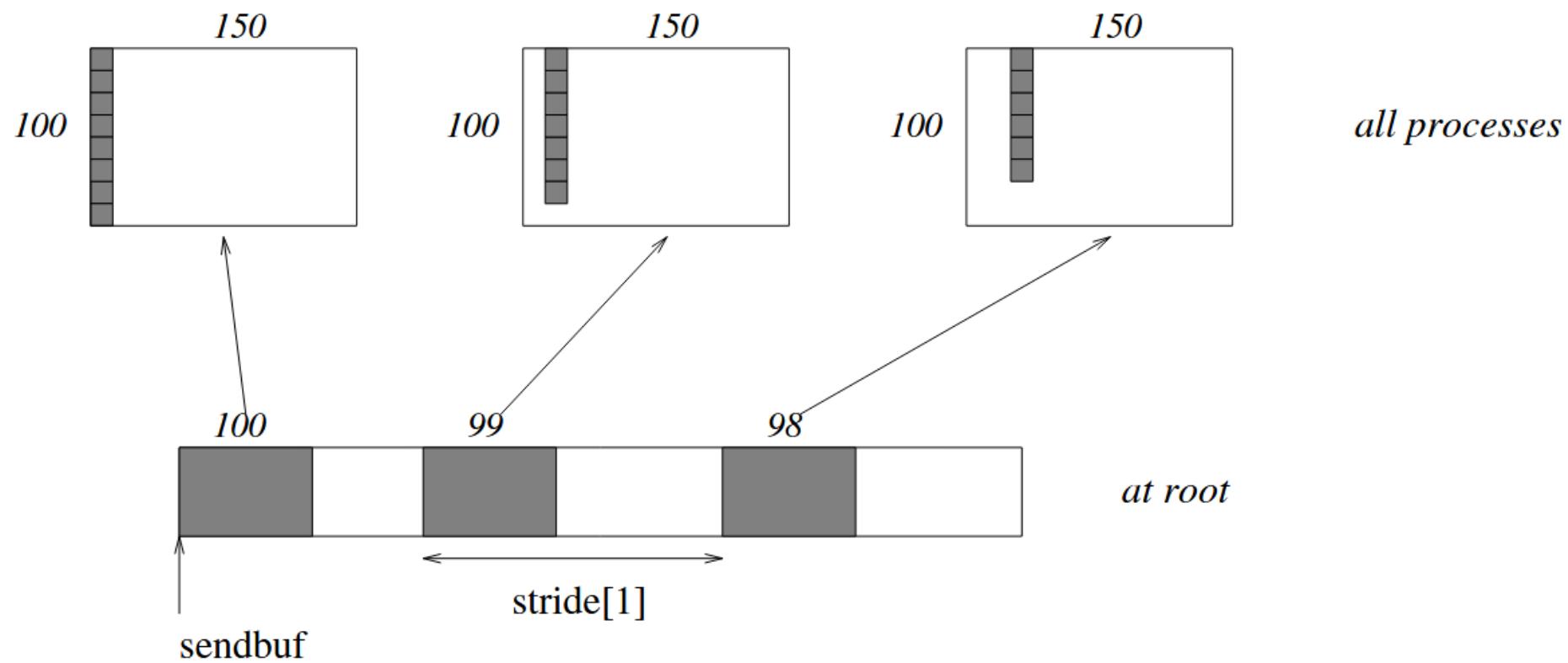
OUT **recvbuf** - адрес начала буфера приема;

IN **recvcount** - число получаемых элементов;

IN **recvtype** - тип получаемых элементов;

IN **root** - номер процесса-отправителя; IN **comm** - коммуникатор.

# Рассылка блоков данных по всем процессам группы



# Рассылка блоков данных по всем процессам группы

```
stride = (int *)malloc(gsize*sizeof(int));
...
/* stride[i] for i = 0 to gsize-1 is set somehow sendbuf comes from elsewhere */
...
displs = (int *)malloc(gsize*sizeof(int));
scounts = (int *)malloc(gsize*sizeof(int));
offset = 0;
for (i=0; i<gsize; ++i) {
    displs[i] = offset;
    offset += stride[i];
    scounts[i] = 100 - i;
}
/* Create datatype for the column we are receiving */
MPI_Type_vector(100-myrank, 1, 150, MPI_INT, &rtype);
MPI_Type_commit(&rtype);
rptr = &recvarray[0][myrank];
MPI_Scatterv(sendbuf, scounts, displs, MPI_INT, rptr, 1, rtype, root, comm);
```

# Рассылка данных от всех процессов всем процессам

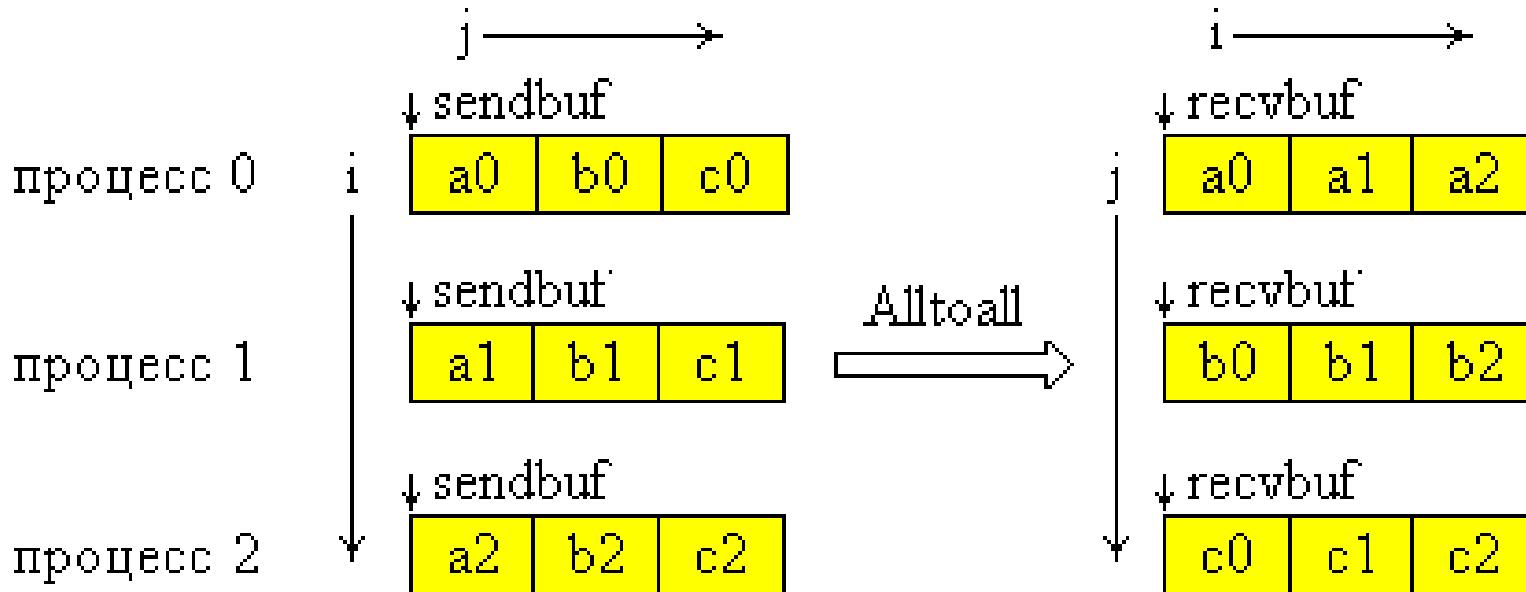
Функция **MPI\_Alltoall** совмещает в себе операции Scatter и Gather и является по сути дела расширением операции Allgather, когда каждый процесс посыпает различные данные разным получателям. Процесс  $i$  посыпает  $j$ -ый блок своего буфера `sendbuf` процессу  $j$ , который помещает его в  $i$ -ый блок своего буфера `recvbuf`. Количество посланных данных должно быть равно количеству полученных данных для каждой пары процессов.

**int MPI\_Alltoall(void\* sendbuf, int sendcount, MPI\_Datatype sendtype,  
void\* recvbuf, int recvcount, MPI\_Datatype recvtype, MPI\_Comm comm)**

IN <code>sendbuf</code>	- адрес начала буфера посылки;
IN <code>sendcount</code>	- число посыпаемых элементов;
IN <code>sendtype</code>	- тип посыпаемых элементов;
OUT <code>recvbuf</code>	- адрес начала буфера приема;
IN <code>recvcount</code>	- число элементов, получаемых от каждого процесса;
IN <code>recvtype</code>	- тип получаемых элементов;
IN <code>comm</code>	- коммуникатор.

**MPI\_Alltoallv** реализует векторный вариант операции Alltoall, допускающий передачу и прием блоков различной длины с более гибким размещением передаваемых и принимаемых данных.

# Рассылка данных от всех процессов всем процессам



# Глобальные вычислительные операции над распределенными данными

*Операцией редукции* называется операция, аргументом которой является вектор, а результатом - скалярная величина, полученная применением некоторой математической операции ко всем компонентам вектора распределенными по процессам коммуникатора.

Операции редукции в MPI представлены в нескольких вариантах:

- с сохранением результата в адресном пространстве одного процесса (MPI\_Reduce).
- с сохранением результата в адресном пространстве всех процессов (MPI\_Allreduce).
- префиксная операция редукции, которая в качестве результата операции возвращает вектор.  $i$ -я компонента этого вектора является результатом редукции первых  $i$  компонент распределенного вектора (MPI\_Scan).
- совмещенная операция Reduce/Scatter (MPI\_Reduce\_scatter).

# Глобальные вычислительные операции над распределенными данными

Функция **MPI\_Reduce** выполняется следующим образом. Операция глобальной редукции, указанная параметром op, выполняется над первыми элементами входного буфера, и результат посыпается в первый элемент буфера приема процесса root. Затем то же самое делается для вторых элементов буфера и т.д.

**int MPI\_Reduce(void\* sendbuf, void\* recvbuf, int count, MPI\_Datatype datatype, MPI\_Op op, int root, MPI\_Comm comm)**

IN sendbuf - адрес начала входного буфера;

OUT recvbuf - адрес начала буфера результатов (используется только в процессе-получателе root);

IN count - число элементов во входном буфере;

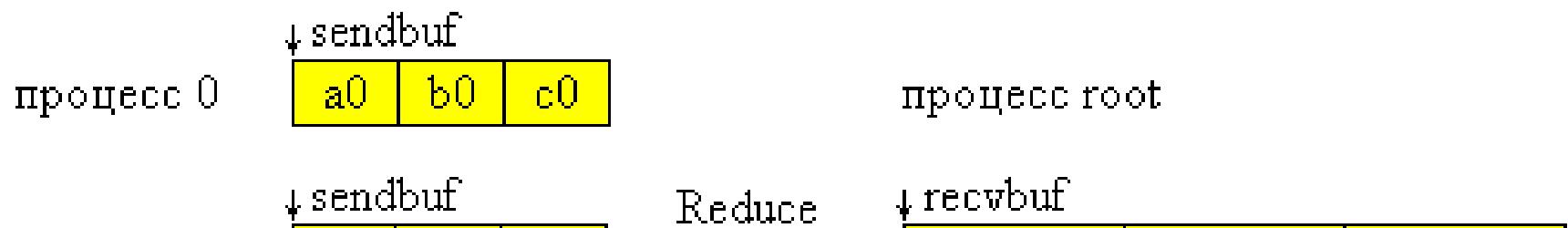
IN datatype - тип элементов во входном буфере;

IN op - операция, по которой выполняется редукция;

IN root - номер процесса-получателя результата операции;

IN comm - коммуникатор.

# Глобальные вычислительные операции над распределенными данными



На данной схеме “+” - ассоциативная операция, которая выполняется над парой operandов типа datatype и возвращает результат того же типа:  
MPI\_MAX, MPI\_MIN, MPI\_SUM,  
MPI\_PROD, MPI\_LAND, MPI\_BAND,  
MPI\_LOR, MPI\_BOR, MPI\_LXOR,  
MPI\_BXOR, MPI\_MAXLOC,  
MPI\_MINLOC

# Глобальные вычислительные операции над распределенными данными

```
MPI_Comm_size ( comm , & groupsize );
MPI_Comm_rank ( comm , & rank );
if ( rank > 0 ) {
    MPI_Recv ( tempbuf , count , datatype , rank -1 ,... )
    User_reduce ( tempbuf , sendbuf , count , datatype );
}
if ( rank < groupsize -1 ) {
    MPI_Send ( sendbuf , count , datatype , rank +1 , ... );
}
/* answer now resides in MPI process groupsize -1 ...now send to root*/
if ( rank == root ) {
    MPI_Irecv ( recvbuf , count , datatype , groupsize -1 ,... , & req );
}
if ( rank == groupsize -1 ) {
    MPI_Send ( sendbuf , count , datatype , root , ... );
}
if ( rank == root ) {
    MPI_Wait ( & req , & status );
}
```

# Cray MPI: параметры по умолчанию

MPI Environment Variable Name	1,000 PEs	10,000 PEs	50,000 PEs	100,000 PEs
<b>MPICH_MAX_SHORT_MSG_SIZE</b> (This size determines whether the message uses the Eager or Randervous protocol)	128,000 Bytes	20,480	4096	2048
<b>MPICH_UNEX_BUFFER_SIZE</b> (The buffer allocated to hold the unexpected Eager data)	60 MB	60 MB	150 MB	260 MB
<b>MPICH_PTL_UNEX_EVENTS</b> (Portals generates two events for each unexpected message received)	20,480 events	22,000	110,000	220,000
<b>MPICH_PTL_OTHER_EVENTS</b> (Portals send-side and expected events)	2048 events	2500	12,500	25,000

# Глобальные вычислительные операции над распределенными данными

```
/* each MPI process has an array of 30 double : ain [30]*/
double ain[30] , aout[30];
int ind[30];
struct {
    double val ;
    int rank ;
} in[30] , out[30];
int i , myrank , root ;
MPI_Comm_rank ( comm , &myrank );
for (i =0; i <30; ++ i) {
    in[i ].val = ain[i ];
    in[i ].rank = myrank ;
}
MPI_Reduce (in , out , 30 , MPI_DOUBLE_INT , MPI_MAXLOC , root , comm );
/* At this point , the answer resides on root MPI process*/
if ( myrank == root ) { /* read ranks out*/
    for (i =0; i <30; ++ i) {
        aout[i] = out[i ].val ;
        ind[i] = out[i ].rank ;
    }
}
```

# Глобальные вычислительные операции над распределенными данными

**MPI\_FLOAT\_INT** float and int

**MPI\_DOUBLE\_INT** double and int

**MPI\_LONG\_INT** long and int

**MPI\_2INT** pair of int

**MPI\_SHORT\_INT** short and int

**MPI\_LONG\_DOUBLE\_INT** long double and int

```
struct mystruct {  
    double val ;  
    uint64_t index ;  
};  
MPI_Datatype dtype ;  
MPI_Type_get_value_index ( MPI_DOUBLE , MPI_UINT64_T , & dtype );  
if ( dtype == MPI_DATATYPE_NULL ) {  
    // Handling for unsupported value - index type  
}
```

# Глобальные вычислительные операции над распределенными данными

```
struct mystruct {  
    short val ;  
    int rank ;  
};  
type [0] = MPI_SHORT ;  
type [1] = MPI_INT ;  
disp [0] = 0;  
disp [1] = offsetof ( struct mystruct , rank );  
block [0] = 1;  
block [1] = 1;  
  
MPI_Type_create_struct (2 , block , disp , type , &MPI_SHORT_INT );  
MPI_Type_commit ( &MPI_SHORT_INT );
```

# Глобальные вычислительные операции над распределенными данными

```
typedef struct {
    double real , imag ;
} Complex ;

/* the user - defined function*/
void myProd ( void * inP , void * inoutP , int * len , MPI_Datatype * dptr )
{
    int i;
    Complex c;
    Complex * in = ( Complex *) inP , * inout = ( Complex *) inoutP ;
    for (i =0; i < * len ; ++ i) {
        c. real = inout -> real *in -> real - inout -> imag *in -> imag ;
        c. imag = inout -> real *in -> imag + inout -> imag *in -> real ;
        * inout = c;
        in++;
        inout++;
    }
}
```

# Глобальные вычислительные операции над распределенными данными

```
...
Complex a [100] , answer [100];
/* each process has an array of 100 Complexes*/

MPI_Op myOp ;
MPI_Datatype ctype ;
/* explain to MPI how type Complex is defined*/

MPI_Type_contiguous (2 , MPI_DOUBLE , & ctype );
MPI_Type_commit (& ctype );

/* create the complex - product user - op*/
MPI_Op_create ( myProd , 1, & myOp );
MPI_Reduce (a , answer , 100 , ctype , myOp , root , comm );
```

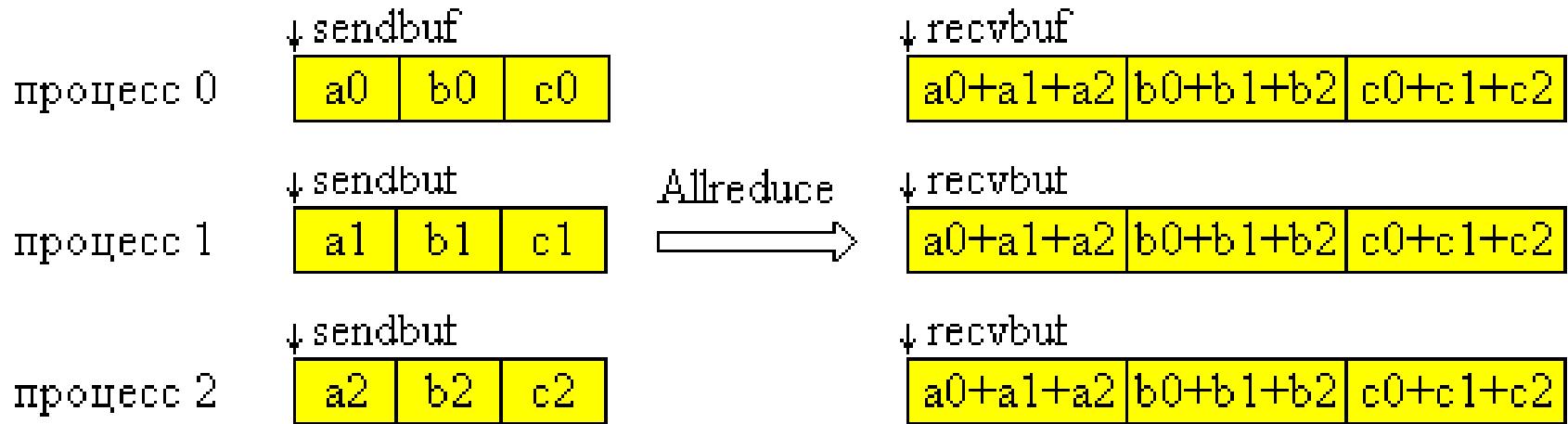
# Глобальные вычислительные операции над распределенными данными

Функция **MPI\_Allreduce** сохраняет результат редукции в адресном пространстве всех процессов, поэтому в списке параметров функции отсутствует идентификатор корневого процесса root. В остальном, набор параметров такой же, как и в предыдущей функции.

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,  
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

- |             |  |
|-------------|--|
| IN sendbuf  | - адрес начала входного буфера;              |
| OUT recvbuf | - адрес начала буфера приема;                |
| IN count    | - число элементов во входном буфере;         |
| IN datatype | - тип элементов во входном буфере;           |
| IN op       | - операция, по которой выполняется редукция; |
| IN comm     | - коммуникатор.                              |

# Глобальные вычислительные операции над распределенными данными



На данной схеме “+” - ассоциативная операция, которая выполняется над парой operandов типа datatype и возвращает результат того же типа:  
MPI\_MAX, MPI\_MIN, MPI\_SUM,  
MPI\_PROD, MPI\_LAND, MPI\_BAND,  
MPI\_LOR, MPI\_BOR, MPI\_LXOR,  
MPI\_BXOR, MPI\_MAXLOC,  
MPI\_MINLOC

# Глобальные вычислительные операции над распределенными данными

Функция **MPI\_Reduce\_scatter** совмещает в себе операции редукции и распределения результата по процессам.

**MPI\_Reduce\_scatter(void\* sendbuf, void\* recvbuf, int \*recvcounts,  
MPI\_Datatype datatype, MPI\_Op op, MPI\_Comm comm)**

IN sendbuf - адрес начала входного буфера;

OUT recvbuf - адрес начала буфера приема;

IN recvcount - массив, в котором задаются размеры блоков, посыпаемых процессам;

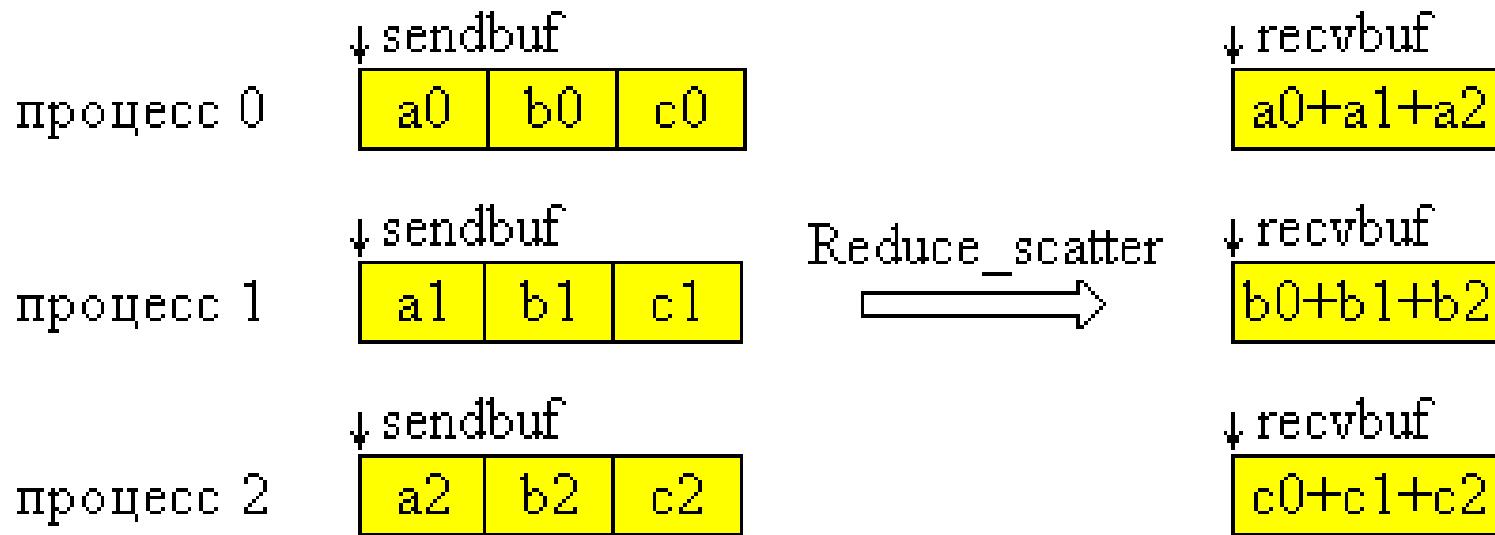
IN datatype - тип элементов во входном буфере;

IN op - операция, по которой выполняется редукция;

IN comm - коммуникатор.

Функция **MPI\_Reduce\_scatter** отличается от **MPI\_Allreduce** тем, что результат операции разрезается на непересекающиеся части по числу процессов в группе,  $i$ -ая часть посыпается  $i$ -ому процессу в его буфер приема. Длины этих частей задает третий параметр, являющийся массивом.

# Глобальные вычислительные операции над распределенными данными



На данной схеме “+” - ассоциативная операция, которая выполняется над парой operandов типа datatype и возвращает результат того же типа:  
MPI\_MAX, MPI\_MIN, MPI\_SUM,  
MPI\_PROD, MPI\_LAND, MPI\_BAND,  
MPI\_LOR, MPI\_BOR, MPI\_LXOR,  
MPI\_BXOR, MPI\_MAXLOC,  
MPI\_MINLOC

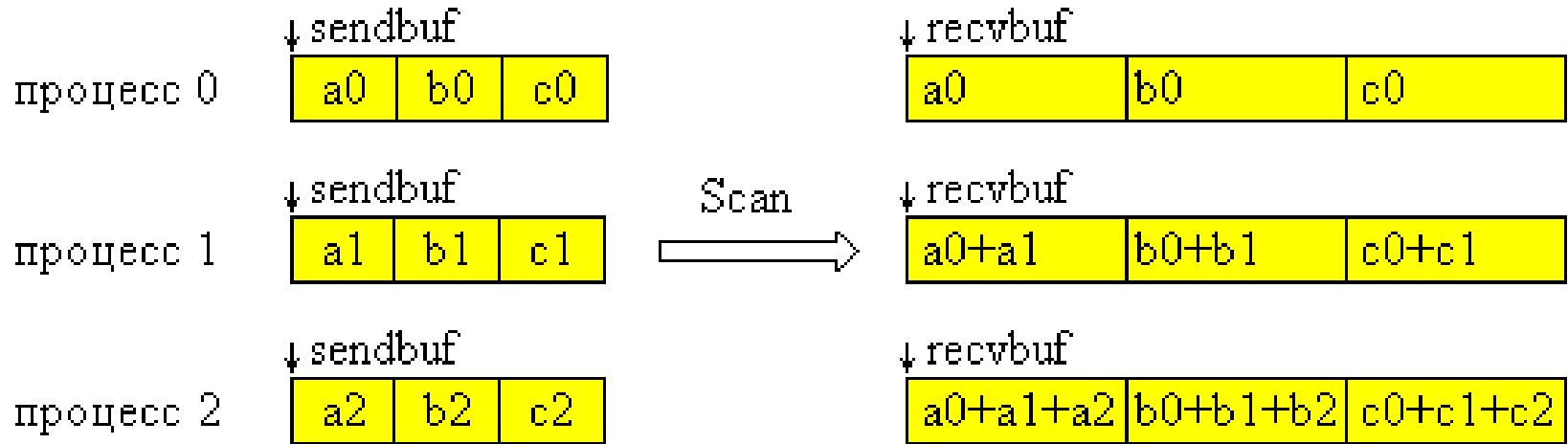
# Глобальные вычислительные операции над распределенными данными

Функция **MPI\_Scan** выполняет префиксную редукцию. Параметры такие же, как в **MPI\_Allreduce**, но получаемые каждым процессом результаты отличаются друг от друга. Операция пересыпает в буфер приема i-го процесса редукцию значений из входных буферов процессов с номерами 0, ... i включительно.

```
int MPI_Scan(void* sendbuf, void* recvbuf, int count,  
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

- |             |   |
|-------------|---|
| IN sendbuf  | - адрес начала входного буфера              |
| OUT recvbuf | - адрес начала буфера приема                |
| IN count    | - число элементов во входном буфере         |
| IN datatype | - тип элементов во входном буфере           |
| IN op       | - операция, по которой выполняется редукция |
| IN comm     | - коммуникатор                              |

# Глобальные вычислительные операции над распределенными данными



На данной схеме “+” - ассоциативная операция, которая выполняется над парой operandов типа datatype и возвращает результат того же типа:  
MPI\_MAX, MPI\_MIN, MPI\_SUM,  
MPI\_PROD, MPI\_LAND, MPI\_BAND,  
MPI\_LOR, MPI\_BOR, MPI\_LXOR,  
MPI\_BXOR, MPI\_MAXLOC,  
MPI\_MINLOC

# Коллективные асинхронные операции

- **MPI\_Ibarrier**
- **MPI\_Ibcast**
- **MPI\_Igather**
- **MPI\_Iscatter**
- **MPI\_Iallgather**
- **MPI\_Ialltoall**

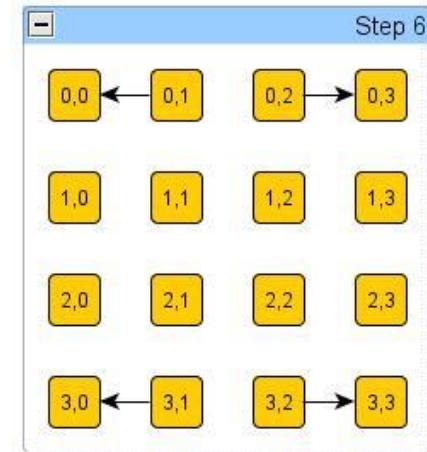
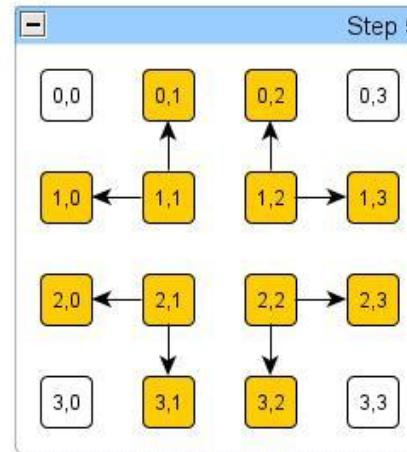
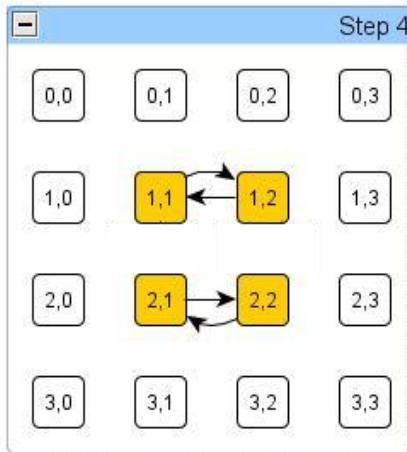
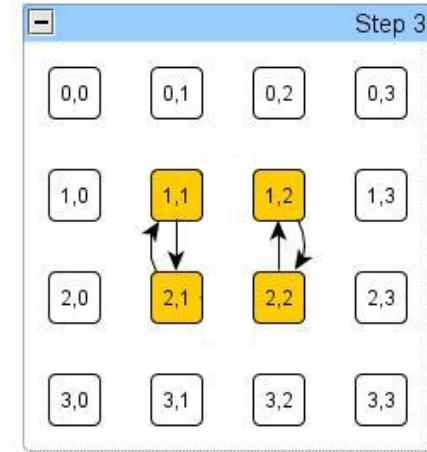
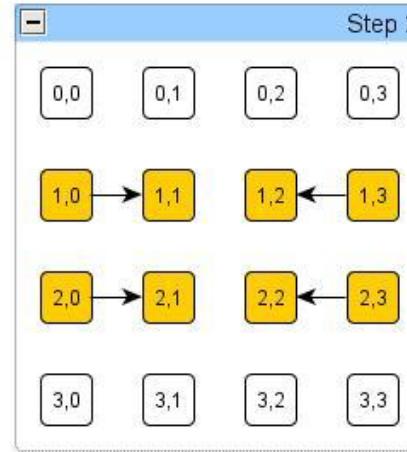
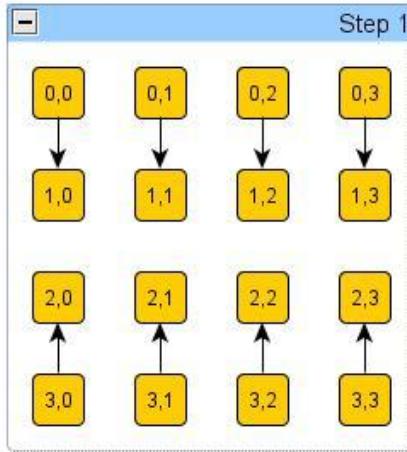
Редукционные операции

- **MPI\_Ireduce**
- **MPI\_Iallreduce**
- **MPI\_Ireduce\_scatter**
- **MPI\_Iscan**
- ....

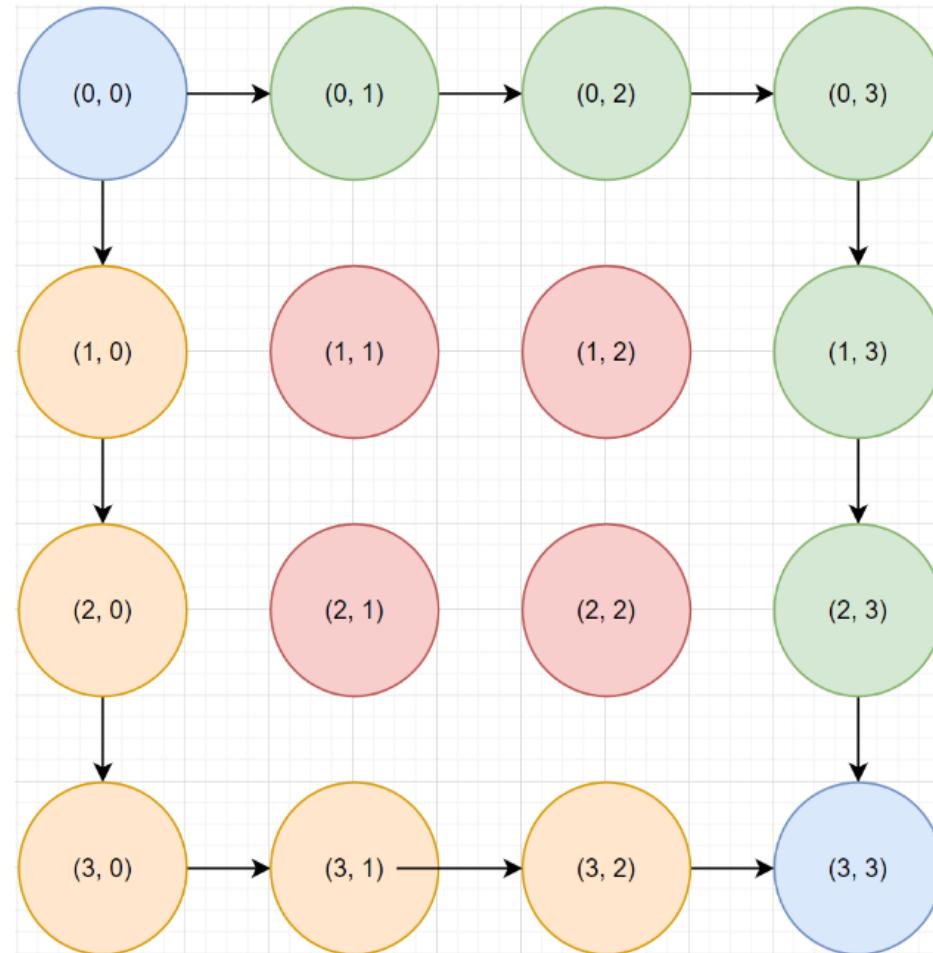
# Коллективные асинхронные операции

```
MPI_Request req;  
switch(rank) {  
    case 0:  
        MPI_Ialltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm, &req);  
        MPI_Wait(&req, MPI_STATUS_IGNORE);  
        break;  
    case 1:  
        MPI_Alltoall(sbuf, scnt, stype, rbuf, rcnt, rtype, comm);  
        break;  
}  
/* erroneous false matching of Alltoall and Ialltoall */
```

# Реализация MPI\_Allreduce



# Передача длинного сообщения



# Partitioned Point-to-Point Communication

```
#include "mpi.h"
#define PARTITIONS 8
#define COUNT 5
int main(int argc, char *argv[])
{
    double message[PARTITIONS*COUNT];
    MPI_Count partitions = PARTITIONS;
    int source = 0, dest = 1, tag = 1, flag = 0;
    int myrank, i;
    int provided;
    MPI_Request request;
    MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
    if (provided < MPI_THREAD_SERIALIZED)
        MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

# Partitioned Point-to-Point Communication

```
if (myrank == 0)
{
    MPI_Psend_init(message, partitions, COUNT, MPI_DOUBLE, dest, tag,
        MPI_COMM_WORLD, MPI_INFO_NULL, &request);
    MPI_Start(&request);
    for(i = 0; i < partitions; ++i)
    {
        /* compute and fill partition #i, then mark ready: */
        MPI_Pready(i, request);
    }
    while(!flag)
    {
        /* do useful work #1 */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* do useful work #2 */
    }
    MPI_Request_free(&request);
}
```

# Partitioned Point-to-Point Communication

```
else if (myrank == 1)
{
    MPI_Precv_init(message, partitions, COUNT, MPI_DOUBLE, source, tag,
                   MPI_COMM_WORLD, MPI_INFO_NULL, &request);
    MPI_Start(&request);
    while(!flag)
    {
        /* do useful work #1 */
        MPI_Test(&request, &flag, MPI_STATUS_IGNORE);
        /* do useful work #2 */
    }
    MPI_Request_free(&request);
}
MPI_Finalize();
return 0;
}
```

# Упаковка и распаковка данных

Входящие в состав сообщения данные могут быть **упакованы** в буфер при помощи функции:

```
int MPI_Pack ( void *data, int count, MPI_Datatype type, void *buf, int  
bufsize, int *bufpos, MPI_Comm comm),
```

где data – буфер памяти с элементами для упаковки,  
count – количество элементов в буфере,  
type – тип данных для упаковываемых элементов,  
buf - буфер памяти для упаковки,  
buflen – размер буфера в байтах,  
bufpos – позиция для начала записи в буфер (в байтах от начала  
буфера),  
comm - коммуникатор для упакованного сообщения.

# Упаковка и распаковка данных

Полученное сообщение может быть **распаковано** при помощи функции:

```
int MPI_Unpack (void *buf, int bufsize, int *bufpos, void *data, int count,  
MPI_Datatype type, MPI_Comm comm)
```

где buf - буфер памяти с упакованными данными,

buflen – размер буфера в байтах,

bufpos – позиция начала данных в буфере (в байтах от начала буфера),

data – буфер памяти для распаковываемых данных,

count – количество элементов в буфере,

type – тип распаковываемых данных,

comm - коммуникатор для упакованного сообщения.

# Упаковка и распаковка данных

Для определения необходимого размера буфера для упаковки может быть использована функция:

```
int MPI_Pack_size (int count, MPI_Datatype type, MPI_Comm comm, int  
*size),
```

которая в параметре **size** указывает необходимый размер буфера для упаковки **count** элементов типа **type**.

После упаковки всех необходимых данных подготовленный буфер может быть использован в функциях передачи данных с указанием типа **MPI\_PACKED**.

# Упаковка и распаковка данных

```
char buff[100];
double x, y;
int position, a[2];
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
if (myrank == 0) { /* Упаковка данных */
    position = 0;
    MPI_Pack(&x, 1, MPI_DOUBLE, buff, 100, &position, MPI_COMM_WORLD);
    MPI_Pack(&y, 1, MPI_DOUBLE, buff, 100, &position, MPI_COMM_WORLD);
    MPI_Pack(a, 2, MPI_INT, buff, 100, &position, MPI_COMM_WORLD);
}
MPI_Bcast(buff, position, MPI_PACKED, 0, MPI_COMM_WORLD);
if (myrank != 0) { /* Распаковка сообщения во всех процессах */
    position = 0;
    MPI_Unpack(buff, 100, &position, &x, 1, MPI_DOUBLE, MPI_COMM_WORLD);
    MPI_Unpack(buff, 100, &position, &y, 1, MPI_DOUBLE, MPI_COMM_WORLD);
    MPI_Unpack(buff, 100, &position, a, 2, MPI_INT, MPI_COMM_WORLD);
}
```

# Point-to-Point Persistent Communication

```
MPI_Request recv_obj, send_obj;  
MPI_Status status;  
//Step 1) Initialize send/request objects  
MPI_Recv_init (buf1, cnt, tp, src, tag, com, &recv_obj);  
MPI_Send_init (buf2, cnt, tp, dst, tag, com, &send_obj);  
for (i=1; i<MAXITER; i++)  
{  
    //Step 2) Use start in place of recv and send  
    //MPI_Irecv (buf1, cnt, tp, src, tag, com, &recv_obj);  
    MPI_Start (&recv_obj);  
    do_work(buf1,buf2);  
    //MPI_Isend (buf2, cnt, tp, dst, tag, com, &send_obj);  
    MPI_Start (&send_obj);  
    //Wait for send to complete  
    MPI_Wait (&send_obj, status);  
    //Wait for receive to finish (no deadlock!)  
    MPI_Wait(&recv_obj, status);  
}  
//Step 3) Clean up the requests  
MPI_Request_free (&recv_obj); MPI_Request_free (&send_obj);
```

# Collective Persistent Communication

```
/* Nonblocking collectives API */
for (i = 0; i < MAXITER; i++) {
    compute(bufA);
    MPI_Ibcast(bufA, ..., rowcomm, &req[0]);
    compute(bufB);
    MPI_Ireduce(bufB, ..., colcomm, &req[1]);
    MPI_Waitall(2, req, ...);
}
```

```
/* Persistent collectives API */
MPI_Bcast_init(bufA, ..., rowcomm, &req[0]);
MPI_Reduce_init(bufB, ..., colcomm, &req[1]);
for (i = 0; i < MAXITER; i++) {
    compute(bufA);
    MPI_Start(req[0]);
    compute(bufB);
    MPI_Start(req[1]);
    MPI_Waitall(2, req, ...);
}
```

# Моделирование сбоя во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int main(int argc, char *argv[])
{
    int rank, size, rc, len;
    char errstr[MPI_MAX_ERROR_STRING];
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    rc = MPI_Barrier(MPI_COMM_WORLD);
    MPI_Error_string(rc, errstr, &len);
    printf("Rank %d / %d: Notified of error %s. Stayin' alive!\n", rank, size, errstr);
    MPI_Finalize();
}
```

# Управление группами процессов

Для получения группы, связанной с существующим коммуникатором, используется функция:

```
int MPI_Comm_group( MPI_Comm comm, MPI_Group *group ).
```

Далее, на основе существующих групп, могут быть созданы новые группы: создание новой группы **newgroup** из существующей группы **oldgroup**, которая будет включать в себя n процессов, ранги которых перечисляются в массиве **ranks**: int

```
MPI_Group_incl(MPI_Group oldgroup,int n, int *ranks,MPI_Group *newgroup),
```

создание новой группы **newgroup** из группы **oldgroup**, которая будет включать в себя n процессов, ранги которых не совпадают с рангами, перечисленными в массиве **ranks**:

```
int MPI_Group_excl(MPI_Group oldgroup,int n, int *ranks,MPI_Group *newgroup).
```

# Управление группами процессов

Для получения новых групп над имеющимися группами процессов могут быть выполнены операции объединения, пересечения и разности:

создание новой группы **newgroup** как объединения групп **group1** и **group2**: int

```
MPI_Group_union(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup);
```

создание новой группы **newgroup** как пересечения групп **group1** и **group2**: int  
**MPI\_Group\_intersection ( MPI\_Group group1, MPI\_Group group2, MPI\_Group \*newgroup ),**

создание новой группы **newgroup** как разности групп **group1** и **group2**: int  
**MPI\_Group\_difference ( MPI\_Group group1, MPI\_Group group2, MPI\_Group \*newgroup ).**

При конструировании групп может оказаться полезной специальная пустая группа **MPI\_COMM\_EMPTY**.

# Управление группами процессов

Ряд функций MPI обеспечивает получение информации о группе процессов:  
получение количества процессов в группе:

**int MPI\_Group\_size ( MPI\_Group group, int \*size ),**

получение ранга текущего процесса в группе:

**int MPI\_Group\_rank ( MPI\_Group group, int \*rank ).**

После завершения использования группа должна быть удалена:

**int MPI\_Group\_free ( MPI\_Group \*group )**

Создание нового коммуникатора из подмножества процессов существующего коммуникатора:

**int MPI\_Comm\_create (MPI\_Comm oldcom, MPI\_Group group, MPI\_Comm \*newcomm)**

# Управление группами процессов

```
MPI_Group WorldGroup, WorkerGroup;  
MPI_Comm Workers;  
int ranks[1];  
ranks[0] = 0;  
// получение группы процессов в MPI_COMM_WORLD  
MPI_Comm_group(MPI_COMM_WORLD, &WorldGroup);  
// создание группы без процесса с рангом 0  
MPI_Group_excl(WorldGroup, 1, ranks, &WorkerGroup);  
// Создание коммуникатора по группе  
MPI_Comm_create(MPI_COMM_WORLD, WorkerGroup, &Workers);  
  
...  
MPI_Group_free(&WorkerGroup);  
MPI_Comm_free(&Workers);
```

# Моделирование сбоя во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
int main(int argc, char *argv[])
{
    int rank, size, rc, len;
    char errstr[MPI_MAX_ERROR_STRING];
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    rc = MPI_Barrier(MPI_COMM_WORLD);
    MPI_Error_string(rc, errstr, &len);
    printf("Rank %d / %d: Notified of error %s. Stayin' alive!\n", rank, size, errstr);
    MPI_Finalize();
}
```

# Моделирование сбоя во время работы MPI-программы

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

static void verbose_errhandler(MPI_Comm* comm, int* err, ...) {
    int rank, size, len;
    char errstr[MPI_MAX_ERROR_STRING];
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Error_string( *err, errstr, &len );
    printf("Rank %d / %d: Notified of error %s\n",
        rank, size, errstr);
}
```

# Моделирование сбоя во время работы MPI-программы

```
int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Errhandler errh;
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_create_errhandler(verbose_errhandler, &errh);
    MPI_Comm_set_errhandler(MPI_COMM_WORLD, errh);
    MPI_Barrier(MPI_COMM_WORLD);
    if( rank == (size-1) ) raise(SIGKILL);
    MPI_Barrier(MPI_COMM_WORLD);
    printf("Rank %d / %d: Stayin' alive!\n", rank, size);
    MPI_Finalize();
}
```

# **Распределенные системы**



**Крюков Виктор Алексеевич**  
д.ф.-м.н., профессор, советник

Института прикладной математики им М.В.Келдыша РАН  
профессор кафедры системного программирования  
факультета вычислительной математики и кибернетики  
Московского университета им. М.В. Ломоносова

**Бахтин Владимир Александрович**  
к.ф.-м.н., ведущий научный сотрудник

Института прикладной математики им М.В.Келдыша РАН  
доцент кафедры системного программирования  
факультета вычислительной математики и кибернетики  
Московского университета им. М.В. Ломоносова

# Введение в предмет

В курсе рассматриваются проблемы создания **распределенных систем** – систем, в которых совокупность независимых компьютеров представляется их пользователям единой объединенной системой.

**Распределенная компьютерная система** – совокупность связанных сетью независимых компьютеров, которая представляется пользователю единым компьютером.

**Распределенная программная система** – совокупность компонентов, взаимодействующих посредством обмена сообщениями.

Основной задачей распределенных систем является облегчение пользователям доступа к удаленным ресурсам и обеспечение их совместного использования.

Обсуждаются способы организации взаимодействия процессов и их доступа к оперативной памяти и файловой системе.

Излагаются принципы обеспечения надежности функционирования распределенных систем.

# Примеры распределенных систем

- сеть рабочих станций,
- кластер ЭВМ,
- система обеспечения банковских операций,
- система резервирования авиабилетов,
- Интернет,
- электронная почта,
- электронная коммерция,
- социальные сети,
- интерактивные игры с множеством игроков, и т.п.

# Примеры распределенных систем



# Черты распределенных систем

- конкурентность
- отсутствие глобальных часов
- независимые отказы

# Тенденции, определяющие развитие РС сегодня

- широкое распространение сетевых технологий
- повсеместное использование компьютеринга в сочетании с желанием поддерживать мобильность пользователей в распределенных системах
- растущий спрос на мультимедийные услуги
- представление распределенных систем как утилиты

# Акцент на совместном использовании ресурсов

- оборудование (принтеры, диски,...)
- данные (файлы, БД)
- сервисы (поиск в интернете, онлайн редактирование, интерактивные игры)

# Ложные предположения, которые делают начинающие разработчики РС

- Сеть надежная
- Сеть безопасна
- Сеть однородная
- Топология не меняется
- Задержка нулевая
- Пропускная способность бесконечна
- Транспортные расходы нулевые
- Есть один администратор

# Проблемы

- гетерогенность
- открытость
- секретность
- масштабируемость
- надежность
- конкурентность
- прозрачность
- качество обслуживания (надежность, секретность, производительность, реактивность, адаптивность)

# Почему создаются распределенные системы? В чем их преимущества перед централизованными ЭВМ?

- Можно достичь такой высокой производительности путем объединения микропроцессоров, которая недостижима в централизованном компьютере.
- Естественная распределенность (банк, поддержка совместной работы группы пользователей).
- Надежность (выход из строя нескольких узлов незначительно снизит производительность).
- Наращиваемость производительности.
  
- Главная причина - наличие огромного количества компьютеров и необходимость совместной работы без ощущения неудобства от географического и физического распределения людей, данных и машин.

# Почему нужно объединять РС в сети?

- Необходимость разделять данные.
- Преимущество разделения дорогих периферийных устройств, уникальных информационных и программных ресурсов.
- Достижение развитых коммуникаций между людьми. Электронная почта во многих случаях удобнее писем, телефонов и факсов.
- Гибкость использования различных ЭВМ, распределение нагрузки.
- Упрощение постепенной модернизации посредством замены компьютеров.

# Принципы построения распределенных систем

- ❑ Прозрачность
- ❑ Гибкость
- ❑ Надежность
- ❑ Эффективность
- ❑ Масштабируемость

# Прозрачность

Прозрачность расположения	Пользователь не должен знать, где расположены ресурсы
Прозрачность миграции	Ресурсы могут перемещаться без изменения их имен
Прозрачность размножения	Пользователь не должен знать, сколько копий существует
Прозрачность конкуренции	Множество пользователей разделяет ресурсы автоматически
Прозрачность параллелизма	Работа может выполняться параллельно без участия пользователя

# Надежность

- ❑ Доступность, устойчивость к ошибкам (fault tolerance)
- ❑ Секретность

# Производительность

- Гранулированность. Мелкозернистый и крупнозернистый параллелизм (fine-grained parallelism, coarse-grained parallelism)
- Устойчивость к ошибкам требует дополнительных накладных расходов

# Масштабируемость

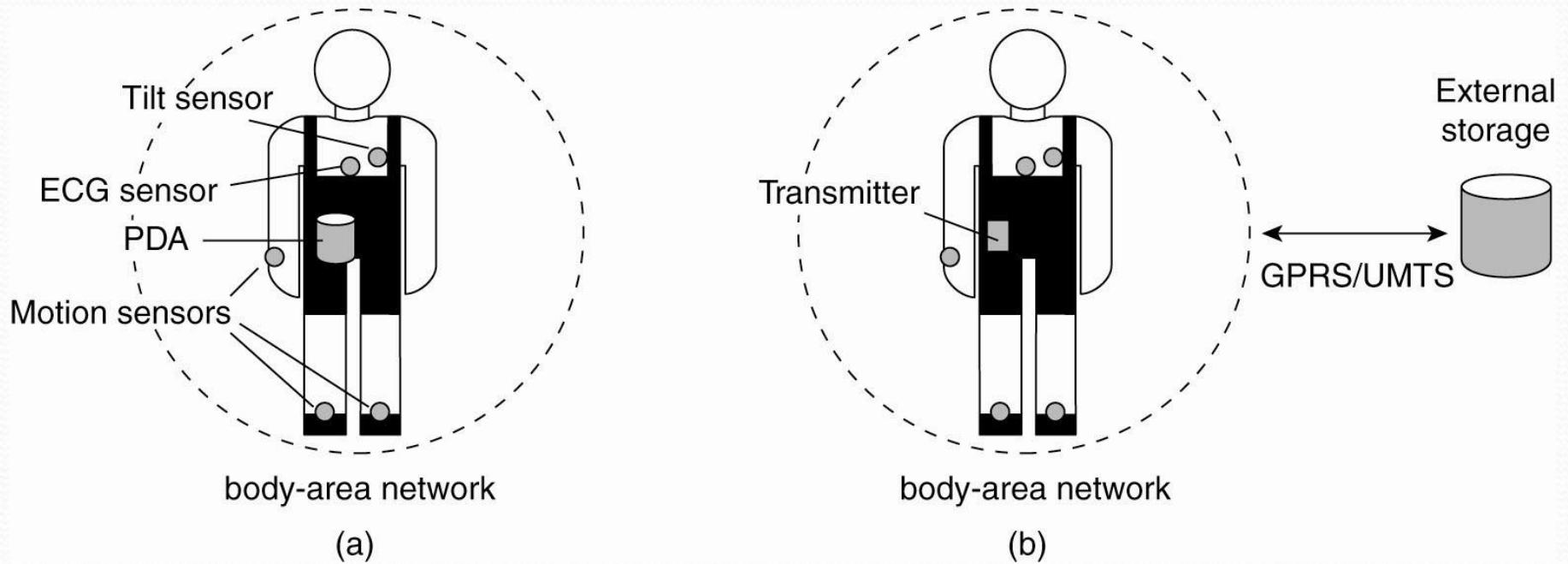
## Плохие решения:

- централизованные компоненты (один почтовый-сервер);
- централизованные таблицы (один телефонный справочник);
- централизованные алгоритмы (маршрутизатор на основе полной информации).

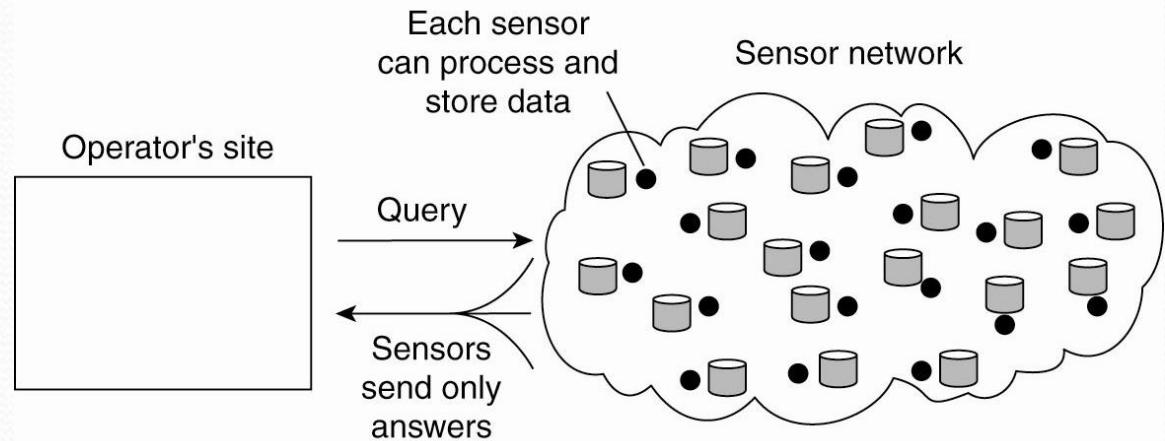
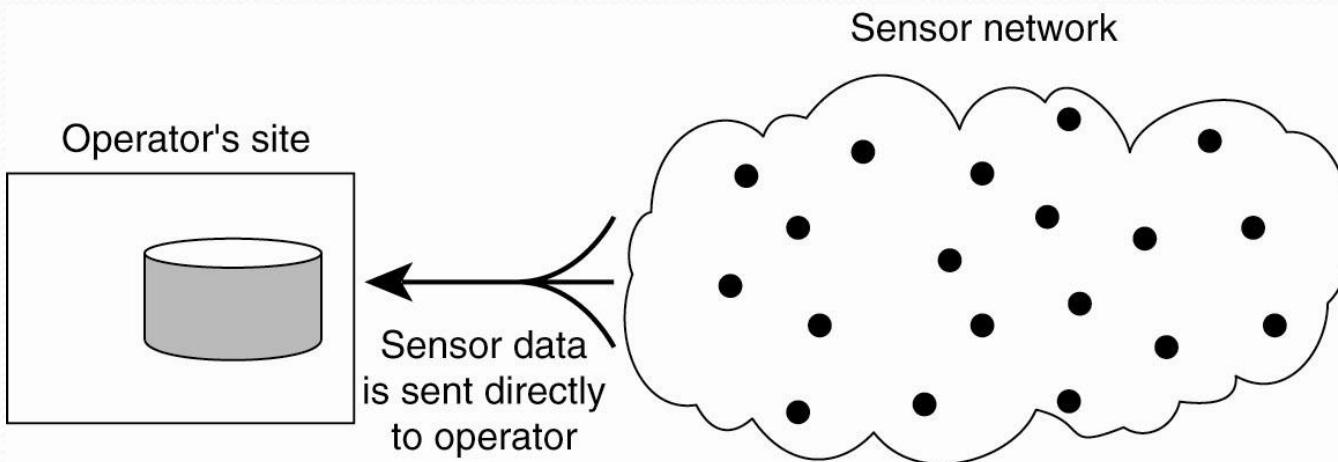
## Только децентрализованные алгоритмы со следующими чертами:

- ни одна машина не имеет полной информации о состоянии системы;
- машины принимают решения на основе только локальной информации;
- выход из строя одной машины не должен приводить к отказу алгоритма;
- не должно быть неявного предположения о существовании глобальных часов.

# Пример. Мониторинг сердечного ритма



# Пример. Мониторинг сердечного ритма



# Основные темы

## □ Достоинства распределенных систем

Прозрачность. Открытость. Масштабируемость.

## □ Процессы

Процессы и потоки выполнения (нити). OpenMP. Многопоточные клиенты и серверы. Взаимное исключение критических интервалов. Алгоритмы Деккера, Петерсона. Семафоры Дейкстры. Механизм событий.

Классические задачи взаимодействия процессов – «производитель-потребитель» и «читатели-писатели».

## □ Коммуникации

Модели взаимодействия. Модель передачи сообщений. MPI. Режимы передачи сообщений. Коллективные операции. Удаленный вызов процедур (Remote Procedure Call).

## □ Синхронизация

Синхронизация времени. Логические часы. Глобальное состояние. Алгоритмы голосования. Взаимное исключение. Распределенные транзакции. Координация процессов.

# Основные темы

- Распределенная разделяемая память (DSM)  
Достоинства разделяемой памяти. Принципы реализации  
распределенной разделяемой памяти. Модели консистентности.  
Страницная DSM. DSM на базе разделяемых переменных.
- Распределенные файловые системы  
Доступ к директориям и файлам. Семантика одновременного  
доступа к одному файлу нескольких процессов. Кэширование и  
размаживание файлов. Примеры - Network File System, HDFS.
- Отказоустойчивость  
Типы отказов. Поломка. Пропуск данных. Ошибка синхронизации.  
Ошибка отклика. Византийские ошибки. Надежная групповая  
рассылка. Протоколы двухфазного и трехфазного подтверждения.  
Фиксация контрольных точек и восстановление после отказа.  
Протоколирование сообщений.
- Примеры распределенных систем  
Проект Hadoop.

# Вопросы по курсу

- В транспьютерной матрице размером  $4 \times 4$ , в каждом узле которой находится один процесс, необходимо переслать очень длинное сообщение (длиной  $L$  байт) из узла с координатами  $(0,0)$  в узел с координатами  $(3,3)$ . Сколько времени потребуется для этого, если передача сообщений точка-точка выполняется в буферизуемом режиме MPI? А сколько времени потребуется при использовании синхронного режима и режима готовности? Время старта равно 100, время передачи байта равно 1 ( $T_s=100, T_b=1$ ). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.
- Все 16 процессов, находящихся на разных ЭВМ сети с шинной организацией (без аппаратных возможностей широковещания), одновременно выдали запрос на вход в критическую секцию. Сколько времени потребуется для прохождения всеми критических секций, если используется древовидный маркерный алгоритм (маркером владеет нулевой процесс). Время старта (время «разгона» после получения доступа кшине для передачи сообщения) равно 100, время передачи байта равно 1 ( $T_s=100, T_b=1$ ). Доступ кшине ЭВМ получают последовательно в порядке выдачи запроса на передачу (при одновременных запросах - в порядке номеров ЭВМ). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

# Вопросы по курсу

- Сколько времени потребует выбор координатора среди 16 процессов, находящихся в узлах транспьютерной матрицы размером 4\*4, если используется круговой алгоритм? Время старта равно 100, время передачи байта равно 1 ( $T_s=100, T_b=1$ ). Процессорные операции, включая чтение из памяти и запись в память считаются бесконечно быстрыми.
- Последовательная консистентность памяти и алгоритм ее реализации в DSM с полным размножением. Сколько времени потребует модификация 10 различных переменных 10-ю процессами (каждый процесс модифицирует одну переменную), находящимися на разных ЭВМ сети с шинной организацией (без аппаратных возможностей широковещания) и одновременно выдавшими запрос на модификацию. Время старта (время «разгона» после получения доступа к шине для передачи сообщения) равно 100, время передачи байта равно 1 ( $T_s=100, T_b=1$ ). Доступ к шине ЭВМ получают последовательно в порядке выдачи запроса на передачу (при одновременных запросах - в порядке номеров ЭВМ). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

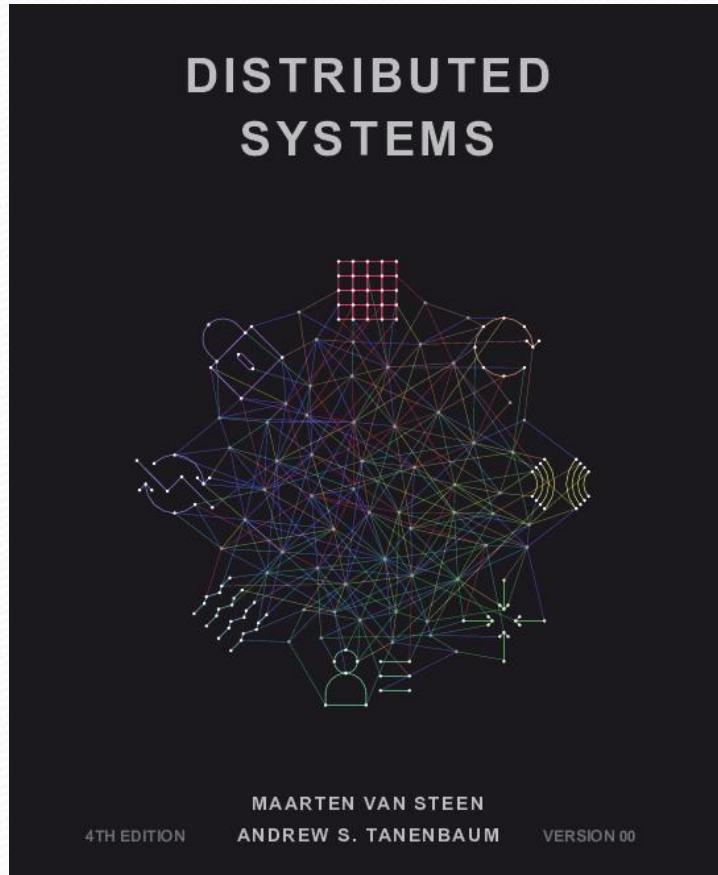
# Вопросы по курсу

- Протоколы голосования. Алгоритмы и применение. Дайте оценку времени выполнения одним процессом 2-х операций записи и 10 операций чтения N байтов информации с файлом, расположенным (размноженным) на остальных 10 ЭВМ сети с шинной организацией (без аппаратных возможностей широковещания). Определите оптимальные значения кворума чтения и кворума записи для N=300. Время старта (время «разгона» после получения доступа к шине для передачи) равно 100, время передачи байта равно 1 ( $T_s=100, T_b=1$ ). Доступ к шине ЭВМ получают последовательно в порядке выдачи запроса (при одновременных запросах - в порядке номеров ЭВМ). Операции с файлами и процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.
- Алгоритм надежных и неделимых широковещательных рассылок сообщений. Дайте оценку времени выполнения одной операции рассылки для сети из 10 ЭВМ с шинной организацией (без аппаратных возможностей широковещания), если отправитель сломался после посылки 5-го сообщения. Время старта (время «разгона» после получения доступа к шине для передачи сообщения) равно 100, время передачи байта равно 1 ( $T_s=100, T_b=1$ ). Доступ к шине ЭВМ получают последовательно в порядке выдачи запроса на передачу (при одновременных запросах - в порядке номеров ЭВМ). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

# Литература

- Э. Таненбаум, М. ван Стейн. Распределенные системы. Принципы и парадигмы.— СПб.: Питер, 2003. — 877 с.: ил. — (Серия «Классика Computer Science») — ISBN 5–272–00053–6.
- В.А. Крюков, В.А. Бахтин. Распределенные системы.  
<http://sp.cs.msu.su> в разделе «Информация».
- Антонов А.С. Технологии параллельного программирования MPI и OpenMP: Учеб. пособие. Предисл.: В.А.Садовничий. - Серия «Суперкомпьютерное образование». М.: Издательство Московского университета, 2012.-344 с.
- Э. Таненбаум. Современные операционные системы. 3-е изд. - СПб.: Питер, 2010. — 1120 с. : ил. — (Серия «Классика Computer Science») — ISBN 978-5-459-00757-2, 978-0136006633.

# Литература



<https://www.distributed-systems.net/>

M. van Steen and A.S. Tanenbaum,  
Distributed Systems, 4th ed., distributed-  
systems.net, 2025

# **Распределенные системы**



**Крюков Виктор Алексеевич**  
д.ф.-м.н., профессор, советник

Института прикладной математики им М.В.Келдыша РАН  
профессор кафедры системного программирования  
факультета вычислительной математики и кибернетики  
Московского университета им. М.В. Ломоносова

**Бахтин Владимир Александрович**  
к.ф.-м.н., ведущий научный сотрудник

Института прикладной математики им М.В.Келдыша РАН  
доцент кафедры системного программирования  
факультета вычислительной математики и кибернетики  
Московского университета им. М.В. Ломоносова

# Введение в предмет

В курсе рассматриваются проблемы создания **распределенных систем** – систем, в которых совокупность независимых компьютеров представляется их пользователям единой объединенной системой.

**Распределенная компьютерная система** – совокупность связанных сетью независимых компьютеров, которая представляется пользователю единым компьютером.

**Распределенная программная система** – совокупность компонентов, взаимодействующих посредством обмена сообщениями.

Основной задачей распределенных систем является облегчение пользователям доступа к удаленным ресурсам и обеспечение их совместного использования.

Обсуждаются способы организации взаимодействия процессов и их доступа к оперативной памяти и файловой системе.

Излагаются принципы обеспечения надежности функционирования распределенных систем.

# Примеры распределенных систем

- сеть рабочих станций,
- кластер ЭВМ,
- система обеспечения банковских операций,
- система резервирования авиабилетов,
- Интернет,
- электронная почта,
- электронная коммерция,
- социальные сети,
- интерактивные игры с множеством игроков, и т.п.

# Примеры распределенных систем



# Черты распределенных систем

- конкурентность
- отсутствие глобальных часов
- независимые отказы

# Тенденции, определяющие развитие РС сегодня

- широкое распространение сетевых технологий
- повсеместное использование компьютеринга в сочетании с желанием поддерживать мобильность пользователей в распределенных системах
- растущий спрос на мультимедийные услуги
- представление распределенных систем как утилиты

# Акцент на совместном использовании ресурсов

- оборудование (принтеры, диски,...)
- данные (файлы, БД)
- сервисы (поиск в интернете, онлайн редактирование, интерактивные игры)

# Ложные предположения, которые делают начинающие разработчики РС

- Сеть надежная
- Сеть безопасна
- Сеть однородная
- Топология не меняется
- Задержка нулевая
- Пропускная способность бесконечна
- Транспортные расходы нулевые
- Есть один администратор

# Проблемы

- гетерогенность
- открытость
- секретность
- масштабируемость
- надежность
- конкурентность
- прозрачность
- качество обслуживания (надежность, секретность, производительность, реактивность, адаптивность)

# Почему создаются распределенные системы? В чем их преимущества перед централизованными ЭВМ?

- Можно достичь такой высокой производительности путем объединения микропроцессоров, которая недостижима в централизованном компьютере.
- Естественная распределенность (банк, поддержка совместной работы группы пользователей).
- Надежность (выход из строя нескольких узлов незначительно снизит производительность).
- Наращиваемость производительности.
  
- Главная причина - наличие огромного количества компьютеров и необходимость совместной работы без ощущения неудобства от географического и физического распределения людей, данных и машин.

# Почему нужно объединять РС в сети?

- Необходимость разделять данные.
- Преимущество разделения дорогих периферийных устройств, уникальных информационных и программных ресурсов.
- Достижение развитых коммуникаций между людьми. Электронная почта во многих случаях удобнее писем, телефонов и факсов.
- Гибкость использования различных ЭВМ, распределение нагрузки.
- Упрощение постепенной модернизации посредством замены компьютеров.

# Принципы построения распределенных систем

- ❑ Прозрачность
- ❑ Гибкость
- ❑ Надежность
- ❑ Эффективность
- ❑ Масштабируемость

# Прозрачность

Прозрачность расположения	Пользователь не должен знать, где расположены ресурсы
Прозрачность миграции	Ресурсы могут перемещаться без изменения их имен
Прозрачность размножения	Пользователь не должен знать, сколько копий существует
Прозрачность конкуренции	Множество пользователей разделяет ресурсы автоматически
Прозрачность параллелизма	Работа может выполняться параллельно без участия пользователя

# Надежность

- ❑ Доступность, устойчивость к ошибкам (fault tolerance)
- ❑ Секретность

# Производительность

- Гранулированность. Мелкозернистый и крупнозернистый параллелизм (fine-grained parallelism, coarse-grained parallelism)
- Устойчивость к ошибкам требует дополнительных накладных расходов

# Масштабируемость

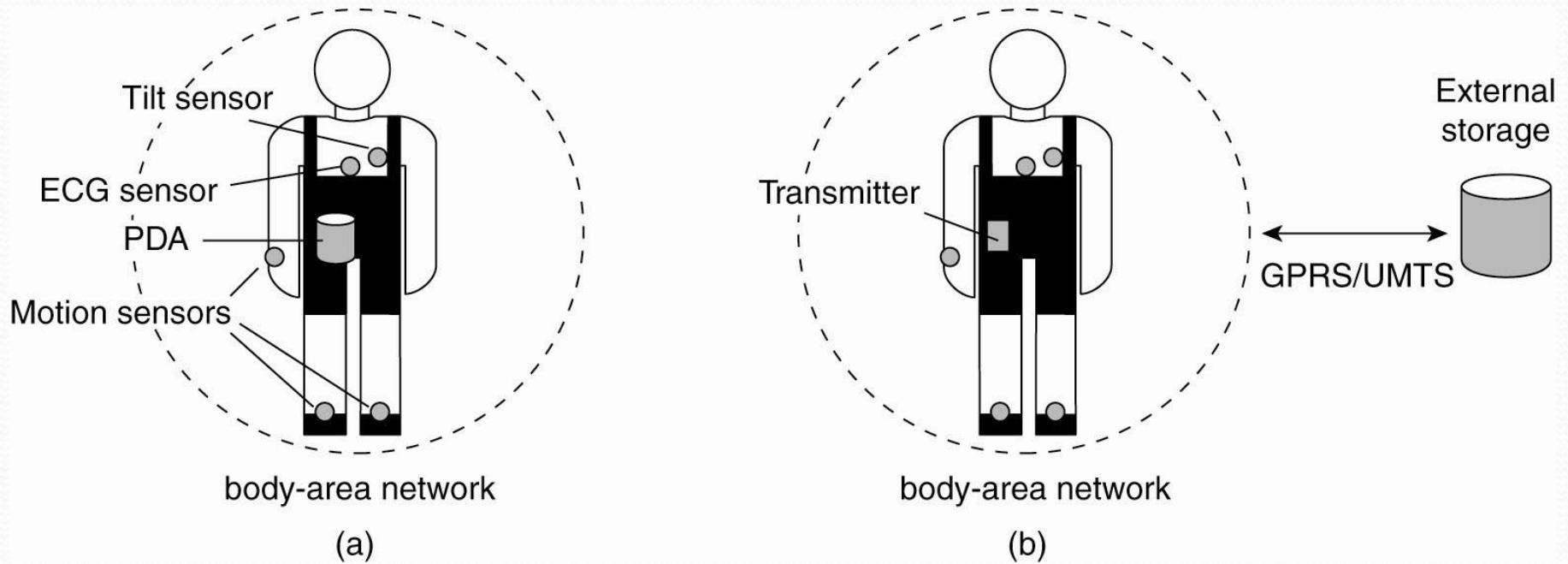
## Плохие решения:

- централизованные компоненты (один почтовый-сервер);
- централизованные таблицы (один телефонный справочник);
- централизованные алгоритмы (маршрутизатор на основе полной информации).

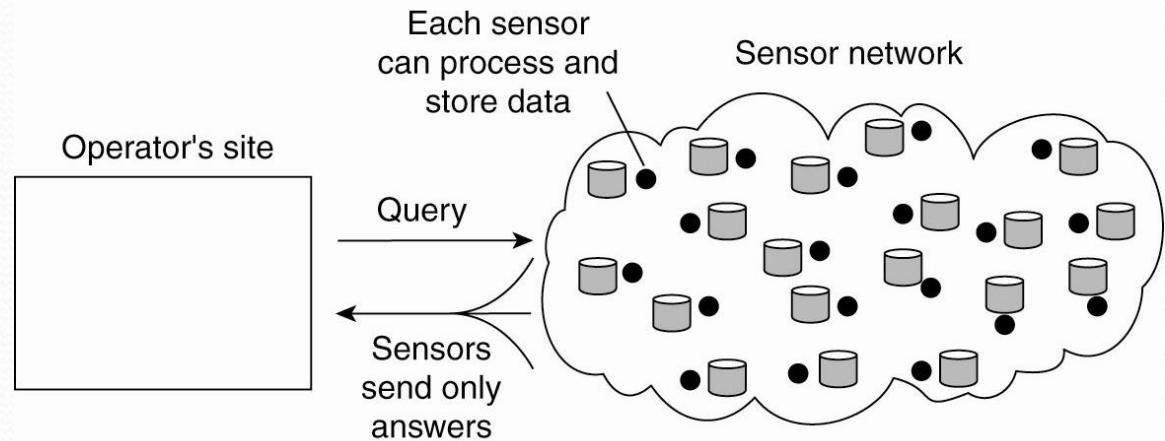
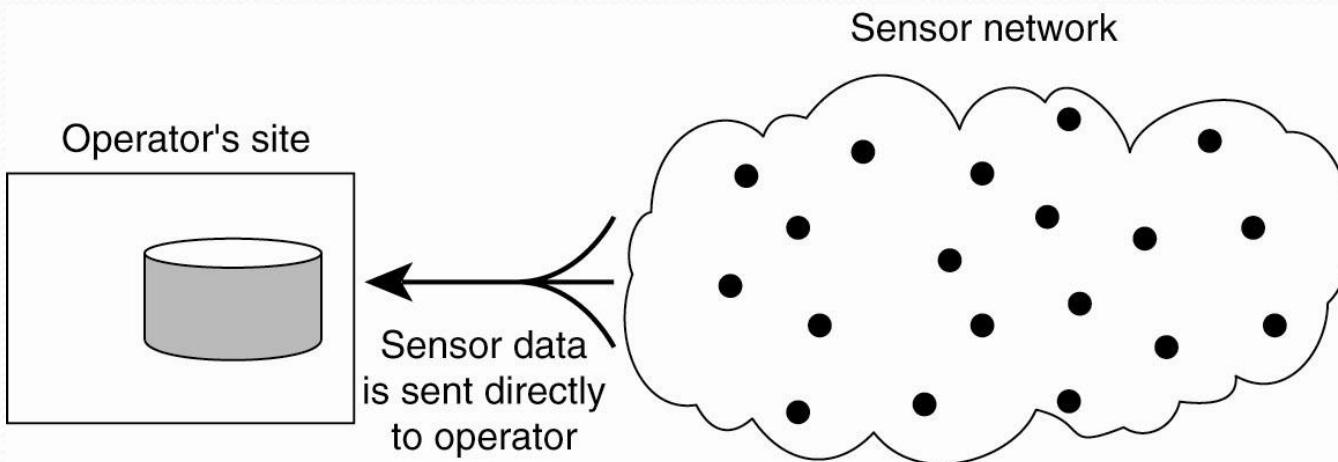
## Только децентрализованные алгоритмы со следующими чертами:

- ни одна машина не имеет полной информации о состоянии системы;
- машины принимают решения на основе только локальной информации;
- выход из строя одной машины не должен приводить к отказу алгоритма;
- не должно быть неявного предположения о существовании глобальных часов.

# Пример. Мониторинг сердечного ритма



# Пример. Мониторинг сердечного ритма



# Основные темы

## □ Достоинства распределенных систем

Прозрачность. Открытость. Масштабируемость.

## □ Процессы

Процессы и потоки выполнения (нити). OpenMP. Многопоточные клиенты и серверы. Взаимное исключение критических интервалов. Алгоритмы Деккера, Петерсона. Семафоры Дейкстры. Механизм событий.

Классические задачи взаимодействия процессов – «производитель-потребитель» и «читатели-писатели».

## □ Коммуникации

Модели взаимодействия. Модель передачи сообщений. MPI. Режимы передачи сообщений. Коллективные операции. Удаленный вызов процедур (Remote Procedure Call).

## □ Синхронизация

Синхронизация времени. Логические часы. Глобальное состояние. Алгоритмы голосования. Взаимное исключение. Распределенные транзакции. Координация процессов.

# Основные темы

- Распределенная разделяемая память (DSM)  
Достоинства разделяемой памяти. Принципы реализации  
распределенной разделяемой памяти. Модели консистентности.  
Страницная DSM. DSM на базе разделяемых переменных.
- Распределенные файловые системы  
Доступ к директориям и файлам. Семантика одновременного  
доступа к одному файлу нескольких процессов. Кэширование и  
размаживание файлов. Примеры - Network File System, HDFS.
- Отказоустойчивость  
Типы отказов. Поломка. Пропуск данных. Ошибка синхронизации.  
Ошибка отклика. Византийские ошибки. Надежная групповая  
рассылка. Протоколы двухфазного и трехфазного подтверждения.  
Фиксация контрольных точек и восстановление после отказа.  
Протоколирование сообщений.
- Примеры распределенных систем  
Проект Hadoop.

# Вопросы по курсу

- В транспьютерной матрице размером  $4 \times 4$ , в каждом узле которой находится один процесс, необходимо переслать очень длинное сообщение (длиной  $L$  байт) из узла с координатами  $(0,0)$  в узел с координатами  $(3,3)$ . Сколько времени потребуется для этого, если передача сообщений точка-точка выполняется в буферизуемом режиме MPI? А сколько времени потребуется при использовании синхронного режима и режима готовности? Время старта равно 100, время передачи байта равно 1 ( $T_s=100, T_b=1$ ). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.
- Все 16 процессов, находящихся на разных ЭВМ сети с шинной организацией (без аппаратных возможностей широковещания), одновременно выдали запрос на вход в критическую секцию. Сколько времени потребуется для прохождения всеми критических секций, если используется древовидный маркерный алгоритм (маркером владеет нулевой процесс). Время старта (время «разгона» после получения доступа кшине для передачи сообщения) равно 100, время передачи байта равно 1 ( $T_s=100, T_b=1$ ). Доступ кшине ЭВМ получают последовательно в порядке выдачи запроса на передачу (при одновременных запросах - в порядке номеров ЭВМ). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

# Вопросы по курсу

- Сколько времени потребует выбор координатора среди 16 процессов, находящихся в узлах транспьютерной матрицы размером 4\*4, если используется круговой алгоритм? Время старта равно 100, время передачи байта равно 1 ( $T_s=100, T_b=1$ ). Процессорные операции, включая чтение из памяти и запись в память считаются бесконечно быстрыми.
- Последовательная консистентность памяти и алгоритм ее реализации в DSM с полным размножением. Сколько времени потребует модификация 10 различных переменных 10-ю процессами (каждый процесс модифицирует одну переменную), находящимися на разных ЭВМ сети с шинной организацией (без аппаратных возможностей широковещания) и одновременно выдавшими запрос на модификацию. Время старта (время «разгона» после получения доступа к шине для передачи сообщения) равно 100, время передачи байта равно 1 ( $T_s=100, T_b=1$ ). Доступ к шине ЭВМ получают последовательно в порядке выдачи запроса на передачу (при одновременных запросах - в порядке номеров ЭВМ). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

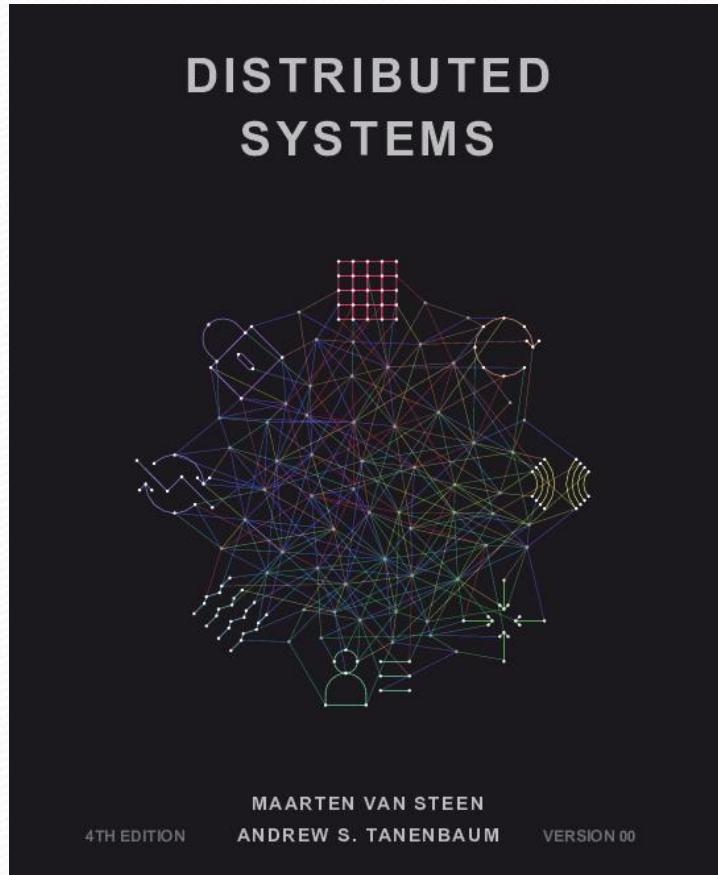
# Вопросы по курсу

- Протоколы голосования. Алгоритмы и применение. Дайте оценку времени выполнения одним процессом 2-х операций записи и 10 операций чтения N байтов информации с файлом, расположенным (размноженным) на остальных 10 ЭВМ сети с шинной организацией (без аппаратных возможностей широковещания). Определите оптимальные значения кворума чтения и кворума записи для N=300. Время старта (время «разгона» после получения доступа к шине для передачи) равно 100, время передачи байта равно 1 ( $T_s=100, T_b=1$ ). Доступ к шине ЭВМ получают последовательно в порядке выдачи запроса (при одновременных запросах - в порядке номеров ЭВМ). Операции с файлами и процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.
- Алгоритм надежных и неделимых широковещательных рассылок сообщений. Дайте оценку времени выполнения одной операции рассылки для сети из 10 ЭВМ с шинной организацией (без аппаратных возможностей широковещания), если отправитель сломался после посылки 5-го сообщения. Время старта (время «разгона» после получения доступа к шине для передачи сообщения) равно 100, время передачи байта равно 1 ( $T_s=100, T_b=1$ ). Доступ к шине ЭВМ получают последовательно в порядке выдачи запроса на передачу (при одновременных запросах - в порядке номеров ЭВМ). Процессорные операции, включая чтение из памяти и запись в память, считаются бесконечно быстрыми.

# Литература

- Э. Таненбаум, М. ван Стейн. Распределенные системы. Принципы и парадигмы.— СПб.: Питер, 2003. — 877 с.: ил. — (Серия «Классика Computer Science») — ISBN 5–272–00053–6.
- В.А. Крюков, В.А. Бахтин. Распределенные системы.  
<http://sp.cs.msu.su> в разделе «Информация».
- Антонов А.С. Технологии параллельного программирования MPI и OpenMP: Учеб. пособие. Предисл.: В.А.Садовничий. - Серия «Суперкомпьютерное образование». М.: Издательство Московского университета, 2012.-344 с.
- Э. Таненбаум. Современные операционные системы. 3-е изд. - СПб.: Питер, 2010. — 1120 с. : ил. — (Серия «Классика Computer Science») — ISBN 978-5-459-00757-2, 978-0136006633.

# Литература



<https://www.distributed-systems.net/>

M. van Steen and A.S. Tanenbaum,  
Distributed Systems, 4th ed., distributed-  
systems.net, 2025

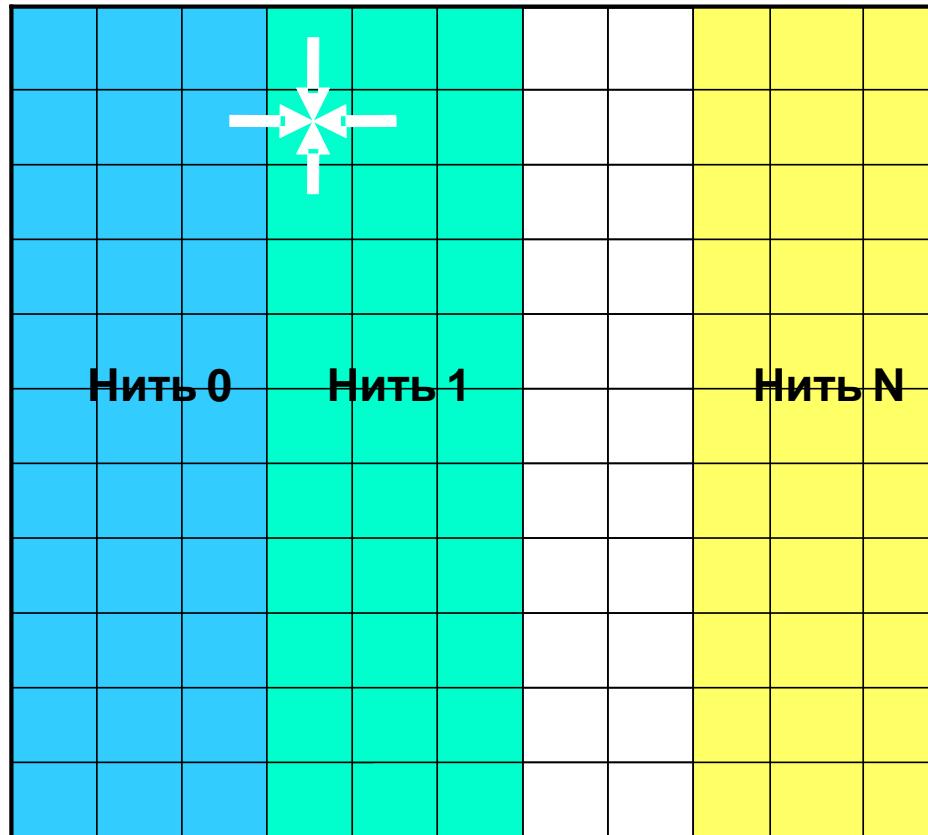
# Синхронизация процессов

# Метод последовательной верхней релаксации (SOR)

```
float A[ L1 ][ L2 ];  
  
for(int i = 1; i < L1-1; i++)  
    for(int j = 1; j < L2-1; j++)  
        A[i][j] = (A[i-1][j] + A[i][j-1] + A[i+1][j] + A[i][j+1]) / 4;
```

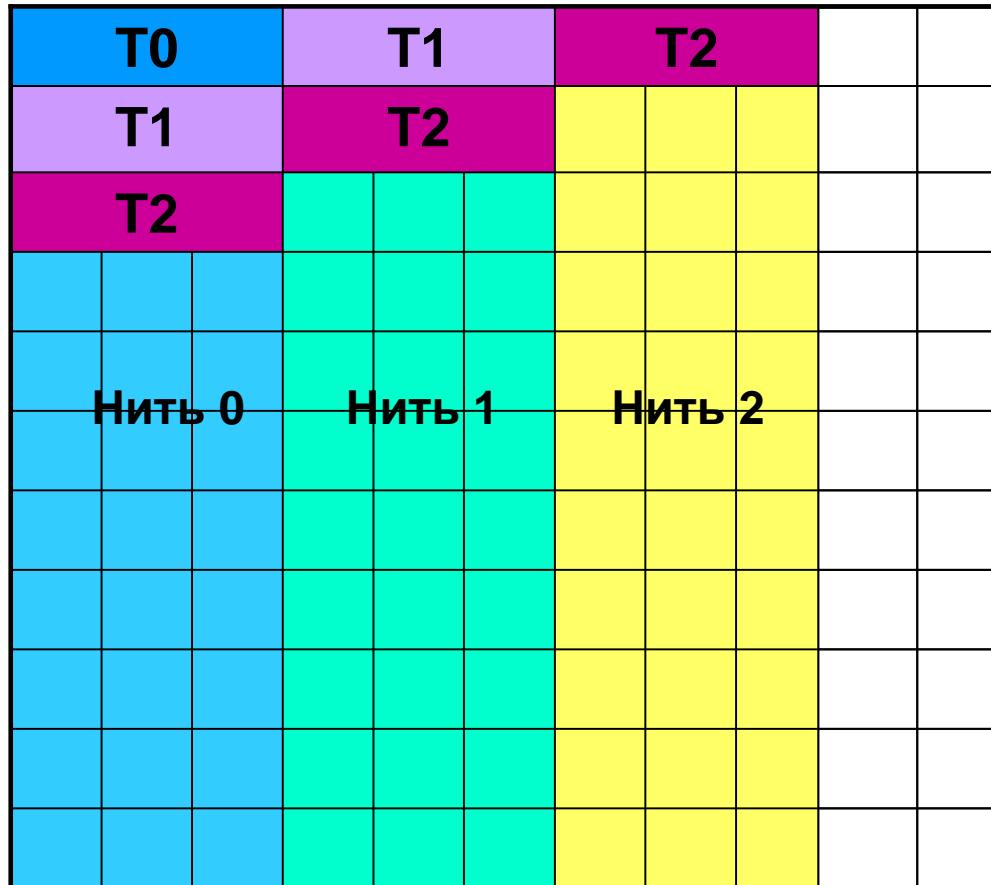
# Метод последовательной верхней релаксации (SOR)

```
for(int i = 1; i < L1-1; i++)  
    for(int j = 1; j < L2-1; j++)  
        A[i][j] = (A[i-1][j] + A[i][j-1] + A[i+1][j] + A[i][j+1]) / 4;
```



# Метод последовательной верхней релаксации (SOR)

```
for(int i = 1; i < L1-1; i++)  
    for(int j = 1; j < L2-1; j++)  
        A[i][j] = (A[i-1][j] + A[i][j-1] + A[i+1][j] + A[i][j+1]) / 4;
```



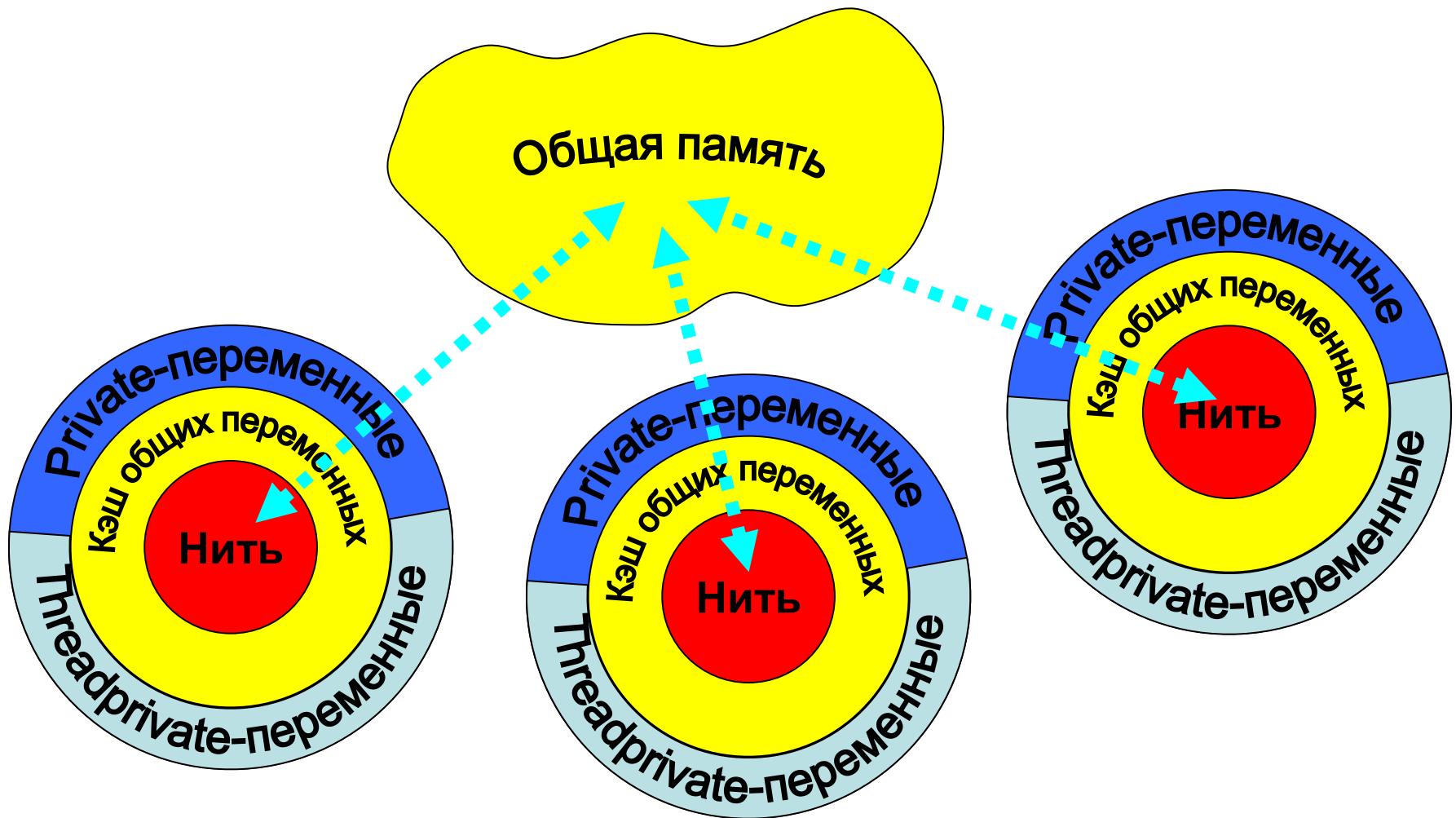
# Метод последовательной верхней релаксации (SOR)

```
int iam, numt, limit;
int sync[NUM_THREADS];
#pragma omp parallel
private(iam,numt,limit)
{
    iam = omp_get_thread_num ();
    numt = omp_get_num_threads ();
    limit=min(numt-1,L2-3);
    isync[iam]=0;
#pragma omp barrier
    for (int i=1; i<L1-1; i++) {
        if ((iam>0) && (iam<=limit)) {
            for (;isync[iam-1]==0;};
            isync[iam-1]=0;
    }
    #pragma omp for schedule(static) nowait
    for (int j=1; j<L2-1; j++) {
        A[i][j]=(A[i-1][j] + A[i][j-1] + A[i+1][j] +
                  A[i][j+1])/4;
    }
    if (iam<limit) {
        isync[iam]=1;
    }
}
}
```

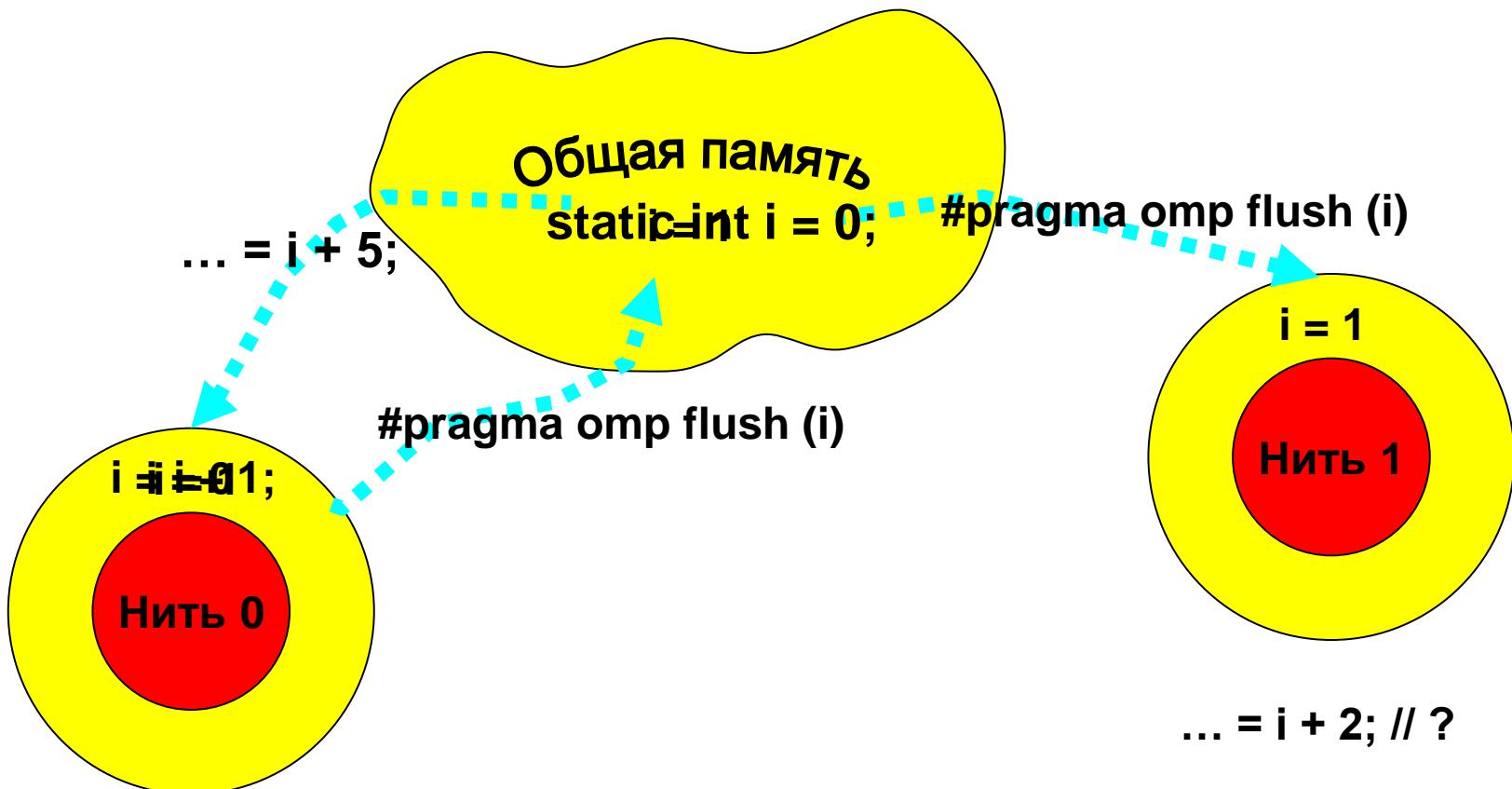
# Метод последовательной верхней релаксации (SOR)

```
int iam, numt, limit;
int sync[NUM_THREADS];
#pragma omp parallel
private(iam,numt,limit)
{
    iam = omp_get_thread_num ();
    numt = omp_get_num_threads ();
    limit=min(numt-1,L2-3);
    isync[iam]=0;
#pragma omp barrier
    for (int i=1; i<L1-1; i++) {
        if ((iam>0) && (iam<=limit)) {
            for (;isync[iam-1]==0;};
            isync[iam-1]=0;
        }
        #pragma omp for schedule(static) nowait
        for (int j=1; j<L2-1; j++) {
            A[i][j]=(A[i-1][j] + A[i][j-1] + A[i+1][j] +
                      A[i][j+1])/4;
        }
        if (iam<limit) {
            for (;isync[iam]==1;};
            isync[iam]=1;
        }
    }
}
```

# Модель памяти в OpenMP



# Модель памяти в OpenMP



# Консистентность памяти в OpenMP

Корректная последовательность работы нитей с переменной:

- Нить0 записывает значение переменной – write (var)
- Нить0 выполняет операцию синхронизации – flush (var)
- Нить1 выполняет операцию синхронизации – flush (var)
- Нить1 читает значение переменной – read (var)

$A = 1$

`flush(A)`

...

`flush(A)`

$\dots = A$

# Консистентность памяти в OpenMP

**#pragma omp flush [(список переменных)]**

По умолчанию все переменные приводятся в консистентное состояние  
**(#pragma omp flush):**

- при барьерной синхронизации;
- при входе и выходе из конструкций **parallel**, **critical** и **ordered**;
- при выходе из конструкций распределения работ (**for**, **single**, **sections**, **workshare**), если не указана клауза **nowait**;
- при вызове **omp\_set\_lock** и **omp\_unset\_lock**;
- при вызове **omp\_test\_lock**, **omp\_set\_nest\_lock**, **omp\_unset\_nest\_lock** и **omp\_test\_nest\_lock**, если изменилось состояние семафора.

При входе и выходе из конструкции **atomic** выполняется **#pragma omp flush(x)**, где x – переменная, изменяемая в конструкции **atomic**.

# Консистентность памяти в OpenMP

1. Если пересечение множеств переменных, указанных в операциях flush, выполняемых различными нитями не пустое, то результат выполнения операций flush будет таким, как если бы эти операции выполнялись в некоторой последовательности (единой для всех нитей).
2. Если пересечение множеств переменных, указанных в операциях flush, выполняемых одной нитью не пустое, то результат выполнения операций flush, будет таким, как если бы эти операции выполнялись в порядке определяемом программой.
3. Если пересечение множеств переменных, указанных в операциях flush, пустое, то операции flush могут выполняться независимо (в любом порядке).

# Метод последовательной верхней релаксации (SOR)

```
int iam, numt, limit;
int sync[NUM_THREADS];
#pragma omp parallel
private(iam,numt,limit)
{
    iam = omp_get_thread_num ();
    numt = omp_get_num_threads ();
    limit=min(numt-1,L2-3);
    isync[iam]=0;
#pragma omp barrier
    for (int i=1; i<L1-1; i++) {
        if ((iam>0) && (iam<=limit)) {
            for (;isync[iam-1]==0;) {
                #pragma omp flush (isync)
            }
            isync[iam-1]=0;
            #pragma omp flush (isync)
        }
    }
    #pragma omp for schedule(static) nowait
    for (int j=1; j<L2-1; j++) {
        A[i][j]=(A[i-1][j] + A[i][j-1] + A[i+1][j] +
        A[i][j+1])/4;
    }
    if (iam<limit) {
        for (;isync[iam]==1;) {
            #pragma omp flush (isync)
        }
        isync[iam]=1;
        #pragma omp flush (isync)
    }
}
```

# Метод последовательной верхней релаксации (SOR)

```
#pragma omp parallel
{
    int iam = omp_get_thread_num ();
    int numt = omp_get_num_threads ();
    for (int newi=1; newi<L1 -1 + numt - 1; newi++) {
        int i = newi - iam;
        #pragma omp for
        for (int j=1; j<L2 - 1; j++) {
            if ((i >= 1) && (i< L1-1)) {
                A[i][j]=(A[i-1][j] + A[i][j-1] + A[i+1][j] + A[i][j+1])/4;
            }
        }
    }
}
```

# Метод последовательной верхней релаксации (SOR)

```
#pragma omp parallel for ordered(2) shared(a)
for (int i=1; i<L1-1; i++)
    for (int j=1; j<L2-1; j++) {
        #pragma omp ordered depend (sink: i - 1, j) depend (sink: i, j - 1)
        A[i][j] = (A[i-1][j]+A[i+1][j]+A[i][j-1]+A[i][j+1])/4;
        #pragma omp ordered depend (source)
    }
/* OpenMP 4.5*/
```

# Механизм событий

События – это переменные, показывающие, что произошли определенные события.

Для объявления события служит оператор

**POST(имя переменной),**

для ожидания события –

**WAIT (имя переменной),**

для чистки (присваивания нулевого значения) - оператор

**CLEAR(имя переменной).**

Варианты реализации - не хранящие информацию (по оператору POST из ожидания выводятся только те процессы, которые уже выдали WAIT), однократно объявляемые (нет оператора чистки).

# Метод последовательной верхней релаксации (SOR)

```
float A[ L1 ][ L2 ];
struct event s[ L1 ][ L2 ];
for ( i = 0; i < L1; i++)
    for ( j = 0; j < L2; j++) { clear( s[ i ][ j ]) };
for ( j = 0; j < L2; j++) { post( s[ 0 ][ j ]) };

.....
.....
parfor ( i = 1; i < L1-1; i++)
    for ( j = 1; j < L2-1; j++)
    {
        wait( s[ i-1 ][ j ]);
        A[ i ][ j ] = (A[ i-1 ][ j ] + A[ i+1 ][ j ] + A[ i ][ j-1 ] + A[ i ][ j+1 ]) / 4;
        post( s[ i ][ j ]);
    }
```

# Метод последовательной верхней релаксации (SOR)

```
float A[ L1 ][ L2 ];
semaphore s[ L1 ][ L2 ];
for ( i = 0; i < L1; i++)
    for ( j = 0; j < L2; j++) { P( s[ i ][ j ] ); }
for ( j = 0; j < L2; j++) { V( s[ 0 ][ j ] ); }

.....
.....
parfor ( i = 1; i < L1-1; i++)
    for ( j = 1; j < L2-1; j++)
    {
        P( s[ i-1 ][ j ] );
        A[ i ][ j ] = (A[ i-1 ][ j ] + A[ i+1 ][ j ] + A[ i ][ j-1 ] + A[ i ][ j+1 ]) / 4;
        V( s[ i ][ j ] );
    }
```

# Метод последовательной верхней релаксации (SOR)

```
float A[ L1 ][ L2 ];
struct event s[ L1 ][ L2 ];
for ( i = 0; i < L1; i++)
    for ( j = 0; j < L2; j++) { clear( s[ i ][ j ]) };
for ( j = 0; j < L2; j++) { post( s[ 0 ][ j ]) };
for ( i = 0; i < L1; i++) { post( s[ i ][ 0 ]) };

.....
.....
parfor ( i = 1; i < L1-1; i++)
    parfor ( j = 1; j < L2-1; j++)
    {
        wait( s[ i-1 ][ j ]);
        wait( s[ i ][ j-1 ]);
        A[ i ][ j ] = (A[ i-1 ][ j ] + A[ i+1 ][ j ] + A[ i ][ j-1 ] + A[ i ][ j+1 ]) / 4;
        post( s[ i ][ j ]);
    }
}
```

# Обмен сообщениями (message passing)

- **Хоар** (Hoare) 1978 год, "Взаимодействующие последовательные процессы". Цели - избавиться от проблем разделения памяти и предложить модель взаимодействия процессов для распределенных систем.

**send** (destination, &message, mszie);  
**receive** ([source], &message, mszie);

- Адресат - процесс.
- Отправитель - может не специфицироваться (любой).
- С буферизацией (почтовые ящики) или нет (рандеву - Ада, Оккам).

Пайпы ОС UNIX - почтовые ящики, заменяют файлы и не хранят границы сообщений (все сообщения объединяются в одно большое, которое можно читать произвольными порциями).

# Производитель-потребитель

```
semaphore mutex = 1;  
semaphore full = 0;  
semaphore empty = N;
```

```
producer()  
{  
    int item;  
    while (TRUE)  
    {  
        produce_item(&item);  
        P(empty);  
        P(mutex);  
        enter_item(item);  
        V(mutex);  
        V(full)  
    }  
}
```

```
consumer()  
{  
    int item;  
    while (TRUE)  
    {  
        P(full);  
        P(mutex);  
        remove_item(&item);  
        V(mutex);  
        V(empty);  
        consume_item(item);  
    }  
}
```

# Производитель-потребитель

```
#define N 100          /* максимальное число сообщений */  
#define msize 4         /* размер сообщения */  
  
typedef int message[msize];
```

```
producer()  
{  
    message m;  
    int item;  
  
    while (TRUE)  
    {  
        produce_item(&item);  
        receive(consumer, &m, msize);  
        build_message(&m, item);  
        send(consumer, &m, msize);  
    }  
}
```

producer() AND consumer()

```
consumer()  
{  
    message m;  
    int item, i;  
  
    for (i = 0; i < N; i++)  
        send (producer, &m, msize);  
    while (TRUE)  
    {  
        receive(producer, &m, msize);  
        extract_item(&m, item);  
        send(producer, &m, msize);  
        consume_item(item);  
    }  
}  
  
/* запустили 2 процесса */
```

# Синхронизация в распределенных системах

# Основные свойства распределенных алгоритмов

Обычно децентрализованные алгоритмы имеют следующие свойства:

- Относящаяся к делу информация распределена среди множества ЭВМ.
- Процессы принимают решение на основе только локальной информации.
- Не должно быть единственной критической точки, выход из строя которой приводил бы к краху алгоритма.
- Не существует общих часов или другого источника точного глобального времени.

Первые три пункта все говорят о недопустимости сбора всей информации для принятия решения в одно место.

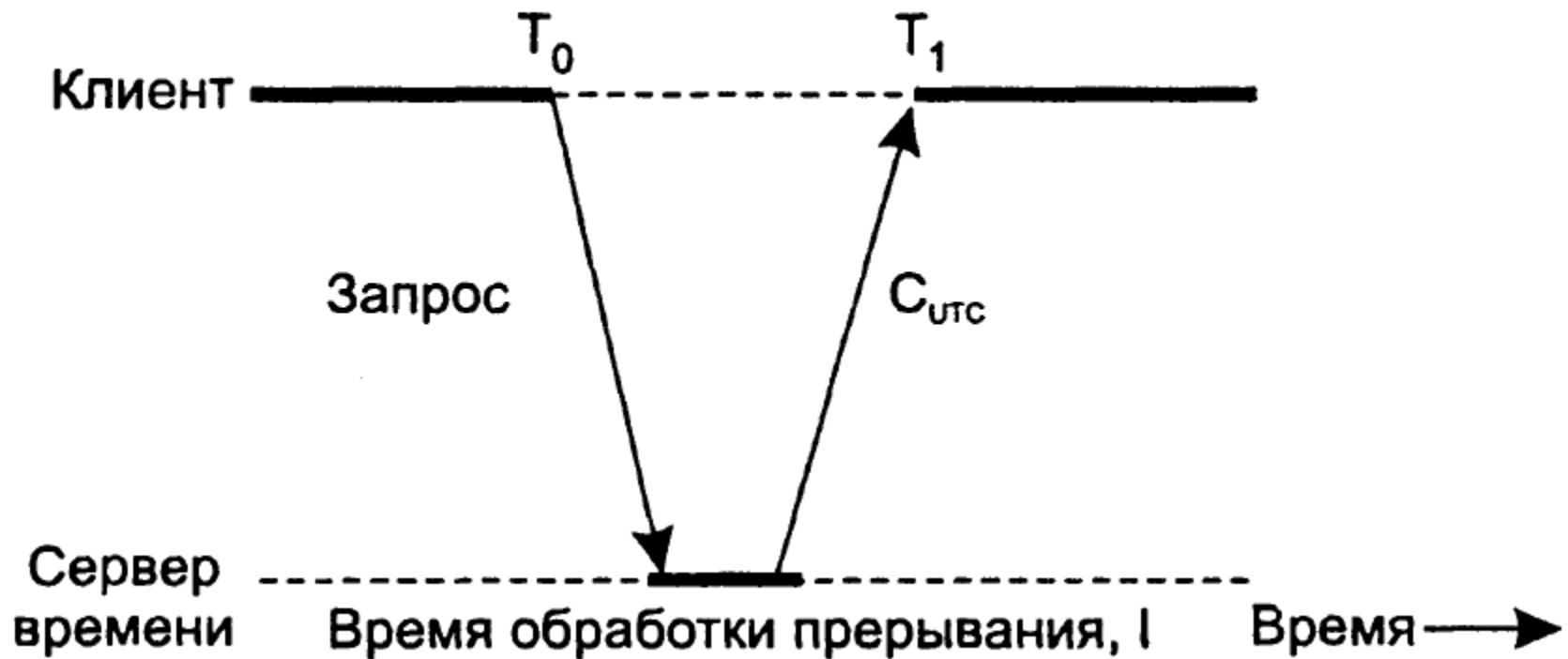
Обеспечение синхронизации без централизации требует подходов, отличных от используемых в традиционных ОС.

Последний пункт также очень важен - в распределенных системах достигнуть согласия относительно времени совсем непросто.

# Синхронизация времени.

## Алгоритм Кристиана

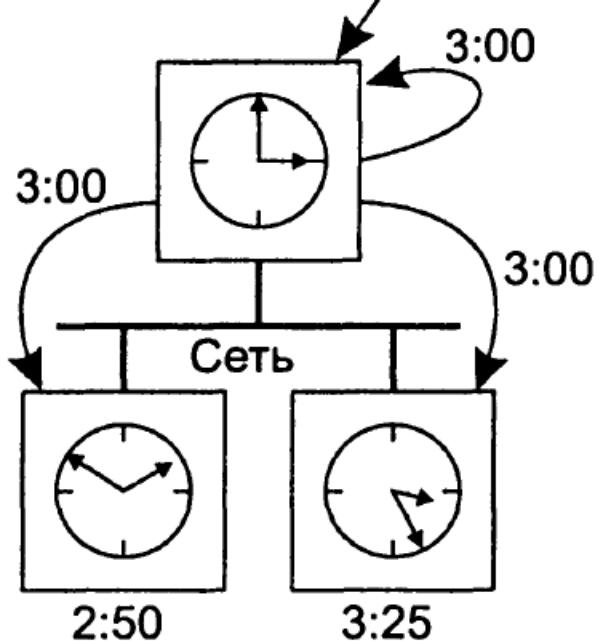
$T_0$  и  $T_1$  отсчитываются по одним и тем же часам



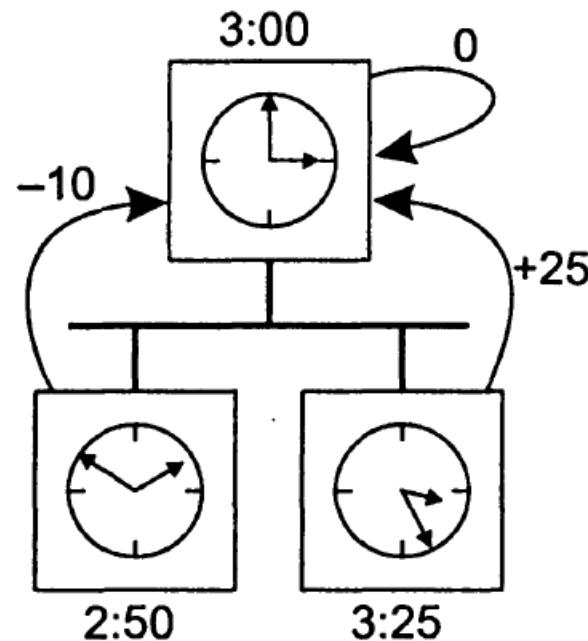
# Синхронизация времени.

## Алгоритм Беркли

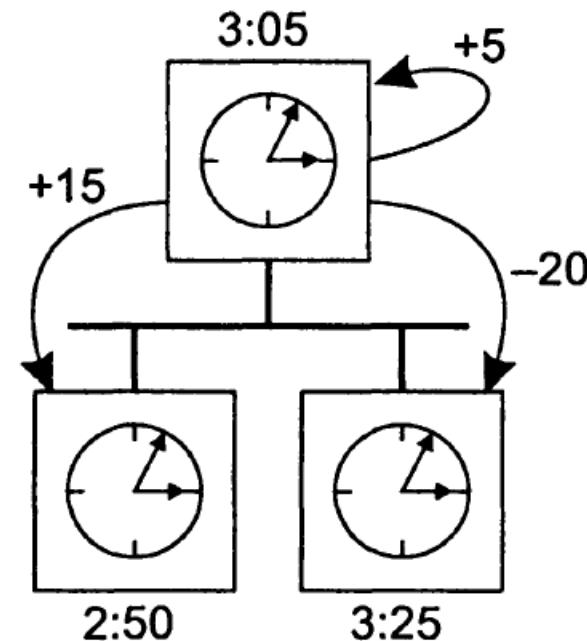
Демон времени



а



б



в

# Логические часы. Lamport

Для синхронизации логических часов Lamport определил отношение «произошло до». Выражение  $a \rightarrow b$  читается как « $a$  произошло до  $b$ » и означает, что все процессы согласны, что сначала произошло событие « $a$ », а затем « $b$ ». Это отношение может в двух случаях быть очевидным:

- 1) Если оба события произошли в одном процессе.
- 2) Если событие « $a$ » есть операция SEND в одном процессе, а событие « $b$ » - прием этого сообщения другим процессом.

Отношение  $\rightarrow$  является транзитивным.

Если два события « $x$ » и « $y$ » случились в различных процессах, которые не обмениваются сообщениями, то отношения  $x \rightarrow y$  и  $y \rightarrow x$  являются неверными, а эти события называют одновременными.

# Логические часы. Lamport

Введем логическое время С таким образом, что  
если  $a \rightarrow b$ , то  $C(a) < C(b)$

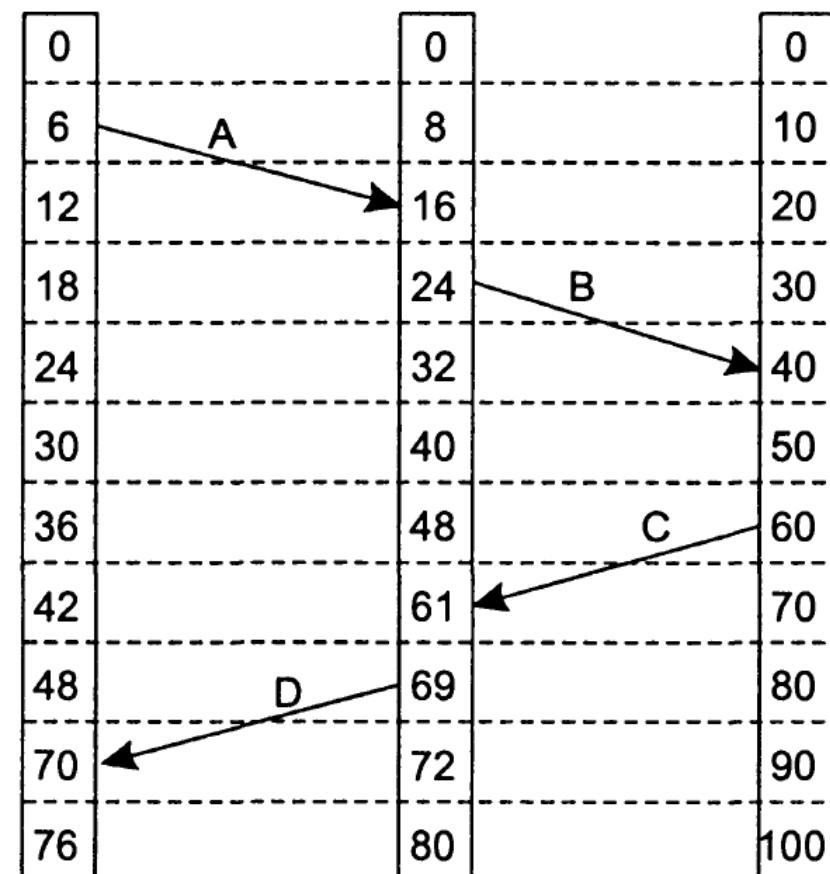
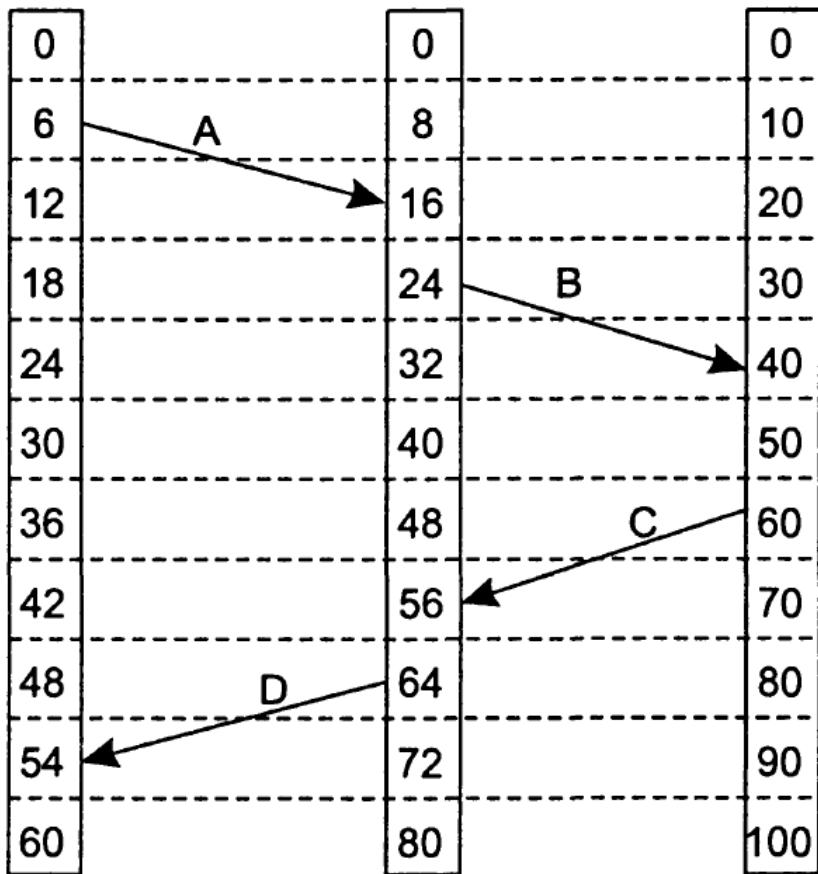
Алгоритм:

- 1) Часы  $C_i$  увеличиваются свое значение с каждым событием в процессе  $P_i$ :  
$$C_i = C_i + d \quad (d > 0, \text{ обычно равно } 1)$$
- 2) Если событие «а» есть посылка сообщения «т» процессом  $P_i$ , тогда в это сообщение вписывается временная метка  $tm=C_i(a)$ .

В момент получения этого сообщения процессом  $P_j$  его время корректируется следующим образом:

$$C_j = \max(C_j, tm+d)$$

# Логические часы. Lamport



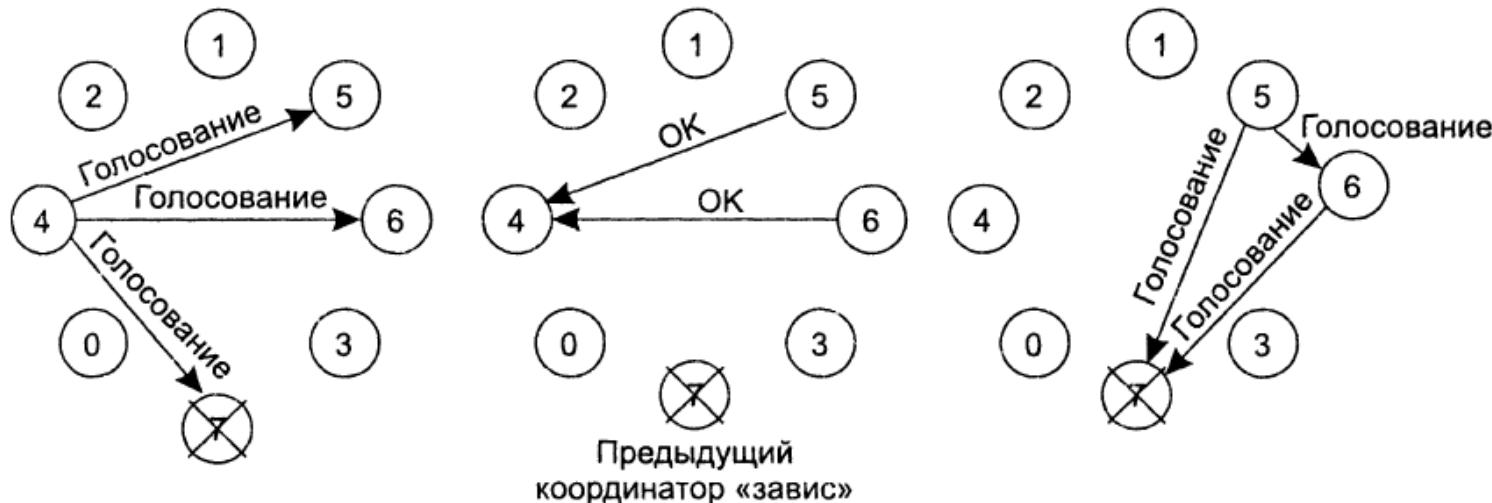
# Выборы координатора.

## Алгоритм задиры

- Если процесс обнаружит, что координатор очень долго не отвечает, то инициирует выборы. Процесс P проводит выборы следующим образом:
- P посылает сообщение «ВЫБОРЫ» всем процессам с большими чем у него номерами.
- Если нет ни одного ответа, то P считается победителем и становится координатором.
- Если один из процессов с большим номером ответит, то он берет на себя проведение выборов. Участие процесса P в выборах заканчивается.
- В любой момент процесс может получить сообщение «ВЫБОРЫ» от одного из коллег с меньшим номером. В этом случае он посылает ответ «OK», чтобы сообщить, что он жив и берет проведение выборов на себя, а затем начинает выборы (если к этому моменту он уже их не вел). Следовательно, все процессы прекратят выборы, кроме одного - нового координатора. Он извещает всех о своей победе и вступлении в должность сообщением «КООРДИНАТОР».
- Если процесс выключился из работы, а затем захотел восстановить свое участие, то он проводит выборы (отсюда и название алгоритма).

# Выборы координатора.

## Алгоритм задиры



а

б

в

г

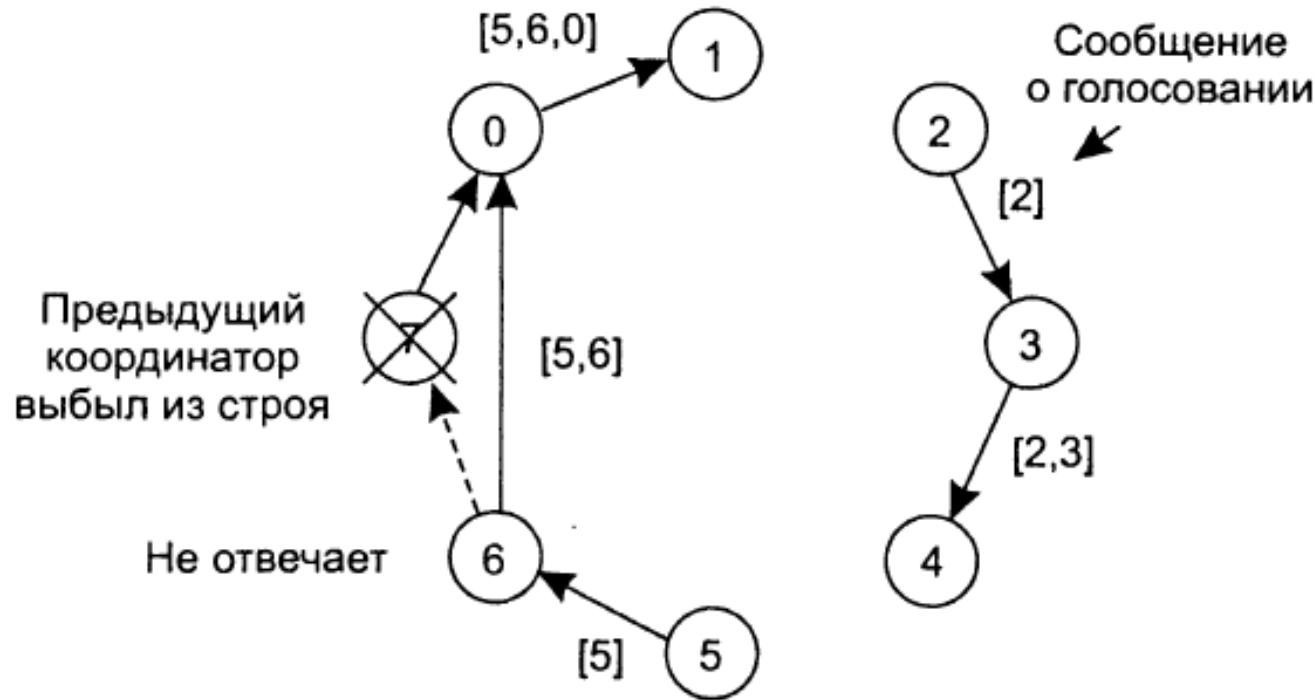
д

# Выборы координатора.

## Круговой алгоритм

- Алгоритм основан на использовании кольца (физического или логического).
- Каждый процесс знает следующего за ним в круговом списке. Когда процесс обнаруживает отсутствие координатора, он посыпает следующему за ним процессу сообщение «ВЫБОРЫ» со своим номером. Если следующий процесс не отвечает, то сообщение посыпается процессу, следующему за ним, и т.д., пока не найдется работающий процесс. Каждый работающий процесс добавляет в список работающих свой номер и переправляет сообщение дальше по кругу.
- Когда процесс обнаружит в списке свой собственный номер (круг пройден), он меняет тип сообщения на «КООРДИНАТОР» и оно проходит по кругу, извещая всех о списке работающих и координаторе (процессе с наибольшим номером в списке).
- После прохождения круга сообщение удаляется.

# Выборы координатора. Круговой алгоритм



# Взаимное исключение.

## Централизованный алгоритм

- Все процессы запрашивают у координатора разрешение на вход в критическую секцию и ждут этого разрешения. Координатор обслуживает запросы в порядке поступления.
- Получив разрешение процесс входит в критическую секцию. При выходе из нее он сообщает об этом координатору.
- Количество сообщений на одно прохождение критической секции - 3.
- Недостатки алгоритма - обычные недостатки централизованного алгоритма (крах координатора или его перегрузка сообщениями).

# Децентрализованный алгоритм на основе временных меток

## *Вход в критическую секцию*

- Когда процесс желает войти в критическую секцию, он посыпает всем процессам сообщение-запрос, содержащее имя критической секции, номер процесса и текущее время.
- После посылки запроса процесс ждет, пока все дадут ему разрешение. После получения от всех разрешения, он входит в критическую секцию.

## *Поведение процесса при приеме запроса*

- Когда процесс получает сообщение-запрос, в зависимости от своего состояния по отношению к указанной критической секции он действует одним из следующих способов.
- Если получатель не находится внутри критической секции и не запрашивал разрешение на вход в нее, то он посыпает отправителю сообщение «OK».
- Если получатель находится внутри критической секции, то он не отвечает, а запоминает запрос.
- Если получатель выдал запрос на вхождение в эту секцию, но еще не вошел в нее, то он сравнивает временные метки своего запроса и чужого. Побеждает тот, чья метка меньше. Если чужой запрос победил, то процесс посыпает сообщение «OK». Если у чужого запроса метка больше, то ответ не посыпается, а чужой запрос запоминается.

# Децентрализованный алгоритм на основе временных меток

## *Выход из критической секции*

- После выхода из секции он посыпает сообщение «OK» всем процессам, запросы от которых он запомнил, а затем стирает все запомненные запросы.
- Количество сообщений на одно прохождение секции -  $2(n-1)$ , где  $n$  - число процессов.
- Кроме того, одна критическая точка заменилась на  $n$  точек (если какой-то процесс перестанет функционировать, то отсутствие разрешения от него всех остановит).
- И, наконец, если в централизованном алгоритме есть опасность перегрузки координатора, то в этом алгоритме перегрузка любого процесса приведет к тем же последствиям.
- Некоторые улучшения алгоритма (например, ждать разрешения не от всех, а от большинства) требуют наличия неделимых широковещательных рассылок сообщений.

Алгоритм носит имя Ricart-Agrawala. Требуется глобальное упорядочение всех событий в системе по времени.

# Алгоритм с круговым маркером

Все процессы составляют логическое кольцо, когда каждый знает, кто следует за ним.

По кольцу циркулирует маркер, дающий право на вход в критическую секцию.

Получив маркер (посредством сообщения точка-точка) процесс либо входит в критическую секцию (если он ждал разрешения) либо переправляет маркер дальше.

После выхода из критической секции маркер переправляется дальше, повторный вход в секцию при том же маркере не разрешается.

# Алгоритм широковещательный маркерный (Suzuki-Kasami)

Маркер содержит:

- очередь запросов;
- массив  $LN[1\dots N]$  с номерами последних удовлетворенных запросов.

## ***Вход в критическую секцию***

- Если процесс  $P_k$ , запрашивающий критическую секцию, не имеет маркера, то он увеличивает порядковый номер своих запросов  $RN_k[k]$  и посыпает широковещательно сообщение «ЗАПРОС», содержащее номер процесса ( $k$ ) и номер запроса ( $S_n = RN_k[k]$ ).
- Процесс  $P_k$  выполняет критическую секцию, если имеет (или когда получит) маркер.

# Алгоритм широковещательный маркерный (Suzuki-Kasami)

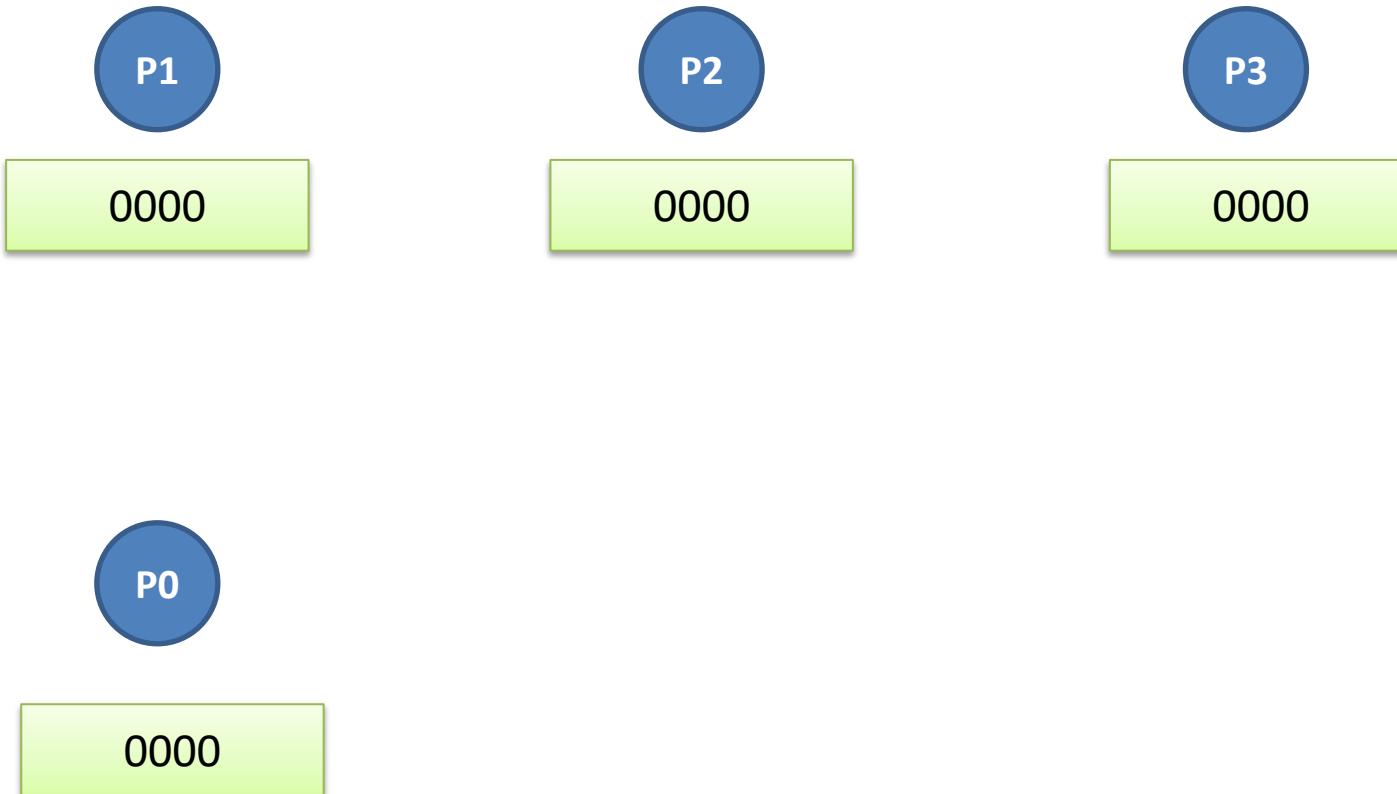
## *Поведение процесса при приеме запроса*

- Когда процесс  $P_j$  получит сообщение-запрос от процесса  $P_k$ , он устанавливает  $RN_j[k] = \max(RN_j[k], S_n)$ .
- Если  $P_j$  имеет свободный маркер, то он его посыпает  $P_k$  только в том случае, когда  $RN_j[k] == LN[k]+1$  (запрос не старый).

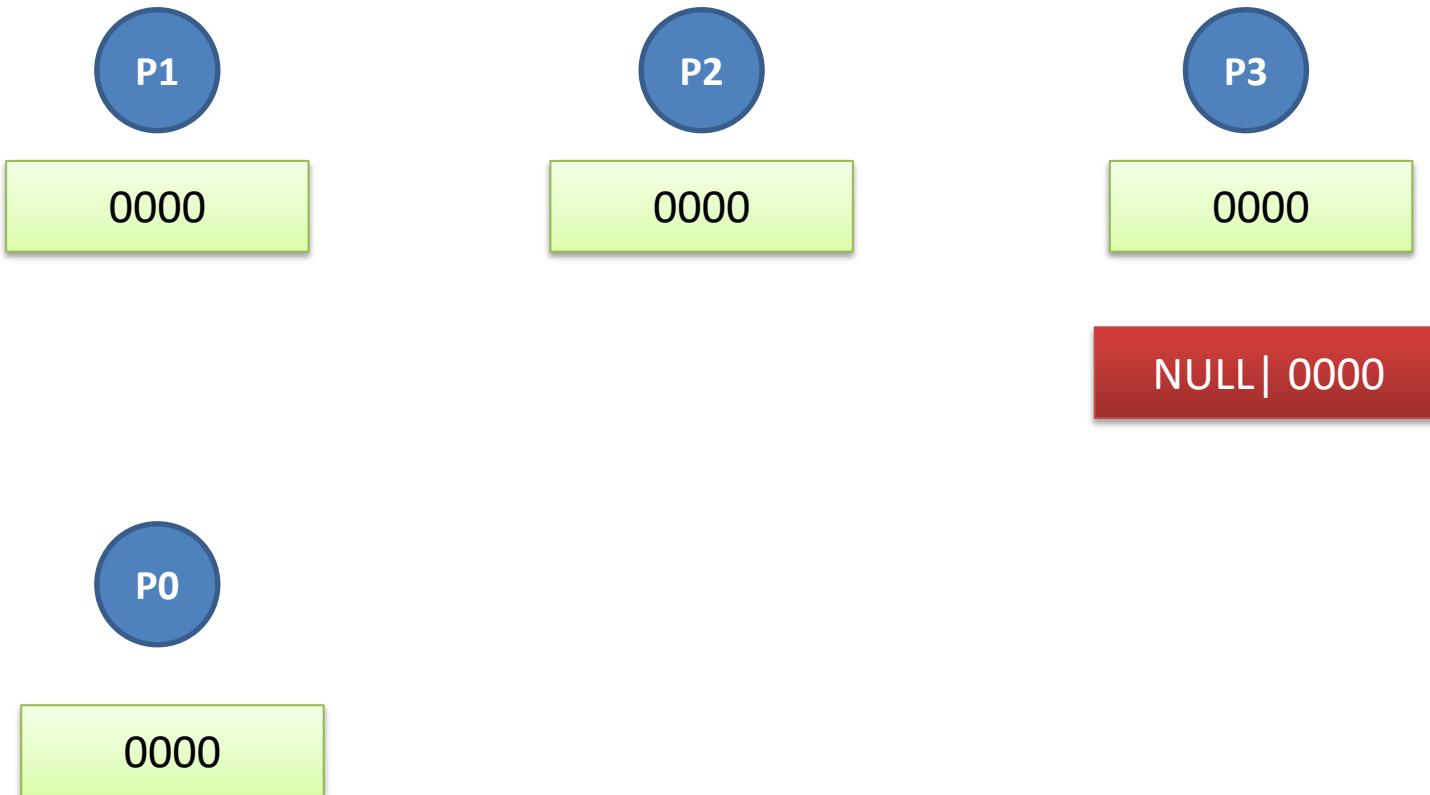
## *Выход из критической секции процесса $P_k$ .*

- Устанавливает  $LN[k]$  в маркере равным  $RN_k[k]$ .
- Для каждого  $P_j$ , для которого  $RN_k[j] = LN[j]+1$ , он добавляет его идентификатор в маркерную очередь запросов (если там его еще нет).
- Если маркерная очередь запросов не пуста, то из нее удаляется первый элемент, а маркер посыпается соответствующему процессу (запрос которого был первым в очереди).

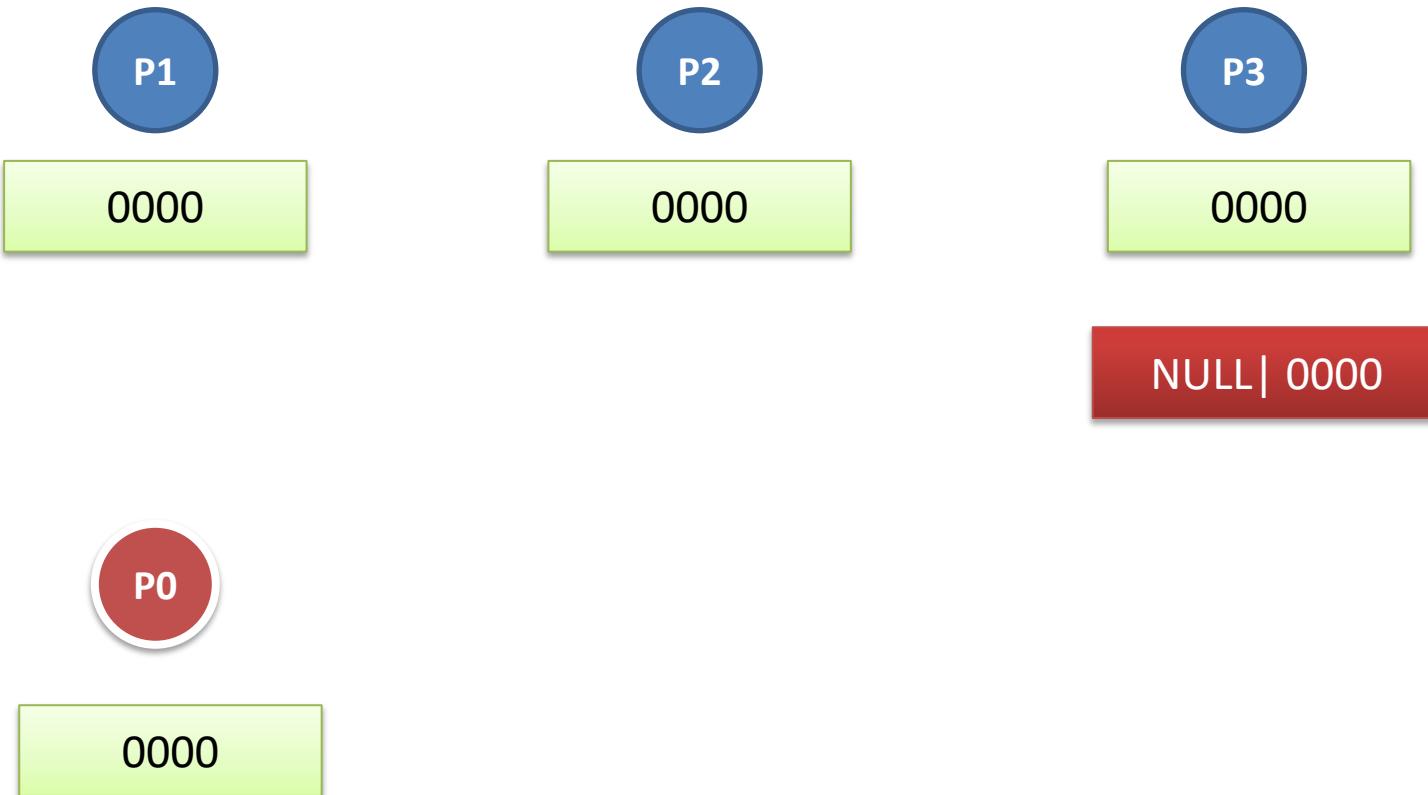
# Алгоритм широковещательный маркерный (Suzuki-Kasami)



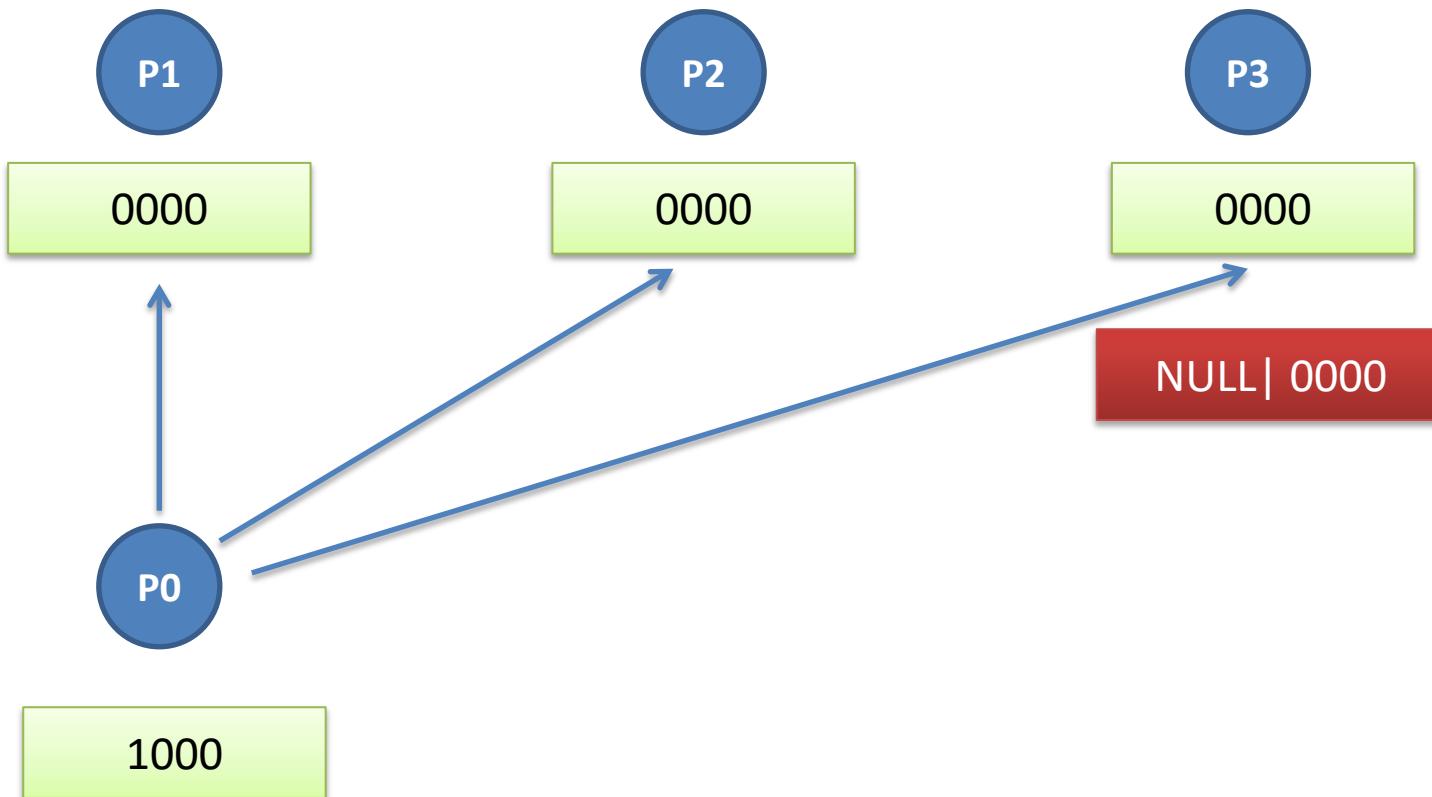
# Алгоритм широковещательный маркерный (Suzuki-Kasami)



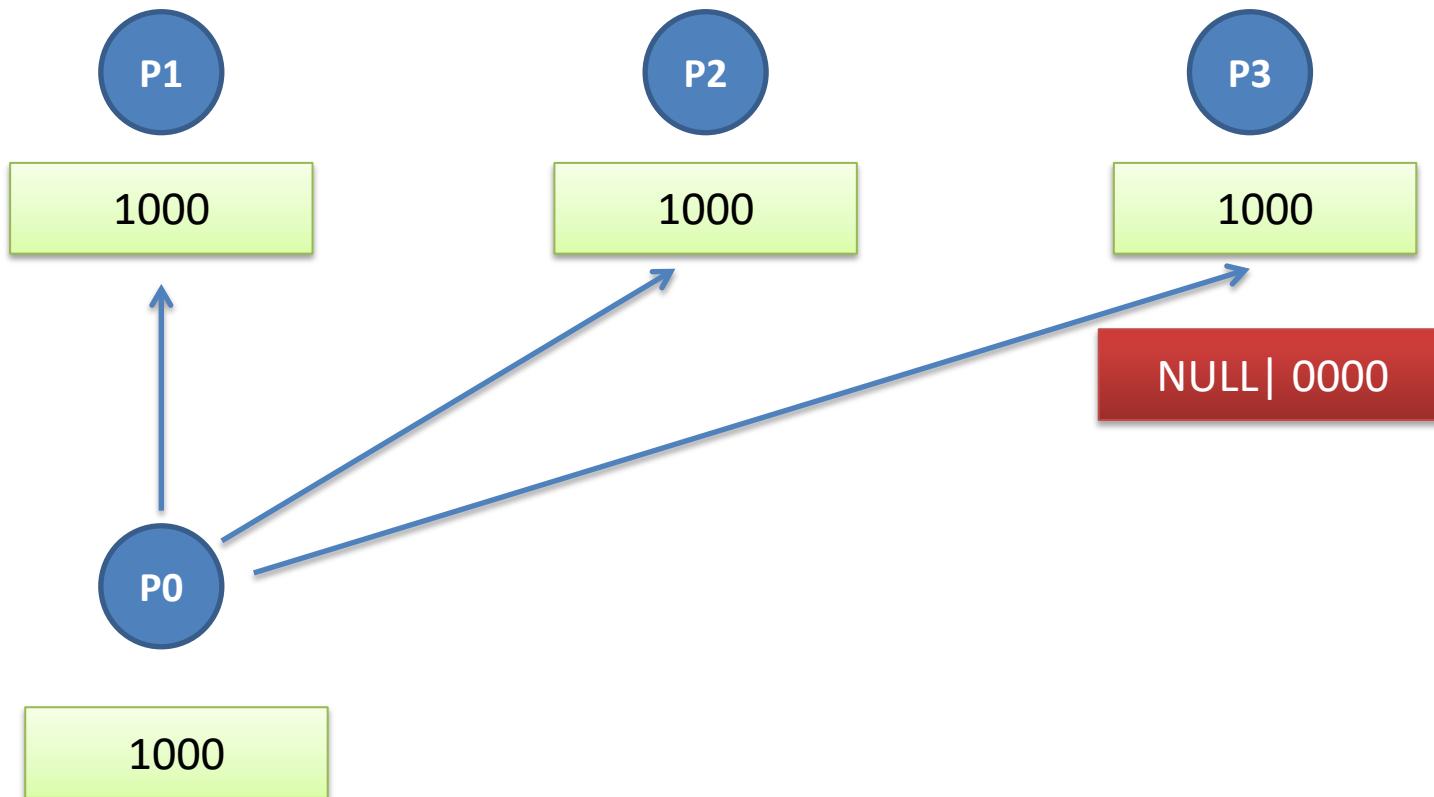
# Алгоритм широковещательный маркерный (Suzuki-Kasami)



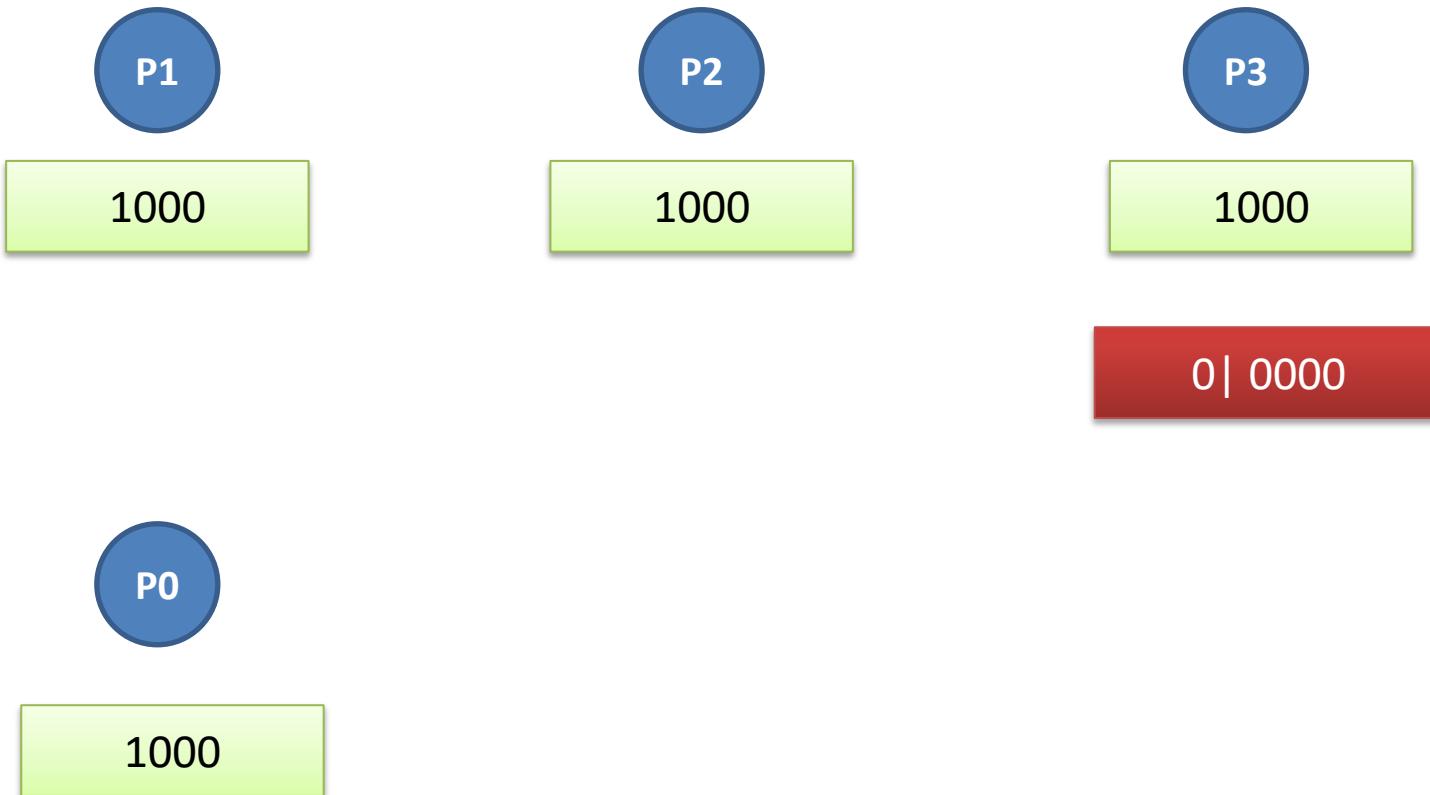
# Алгоритм широковещательный маркерный (Suzuki-Kasami)



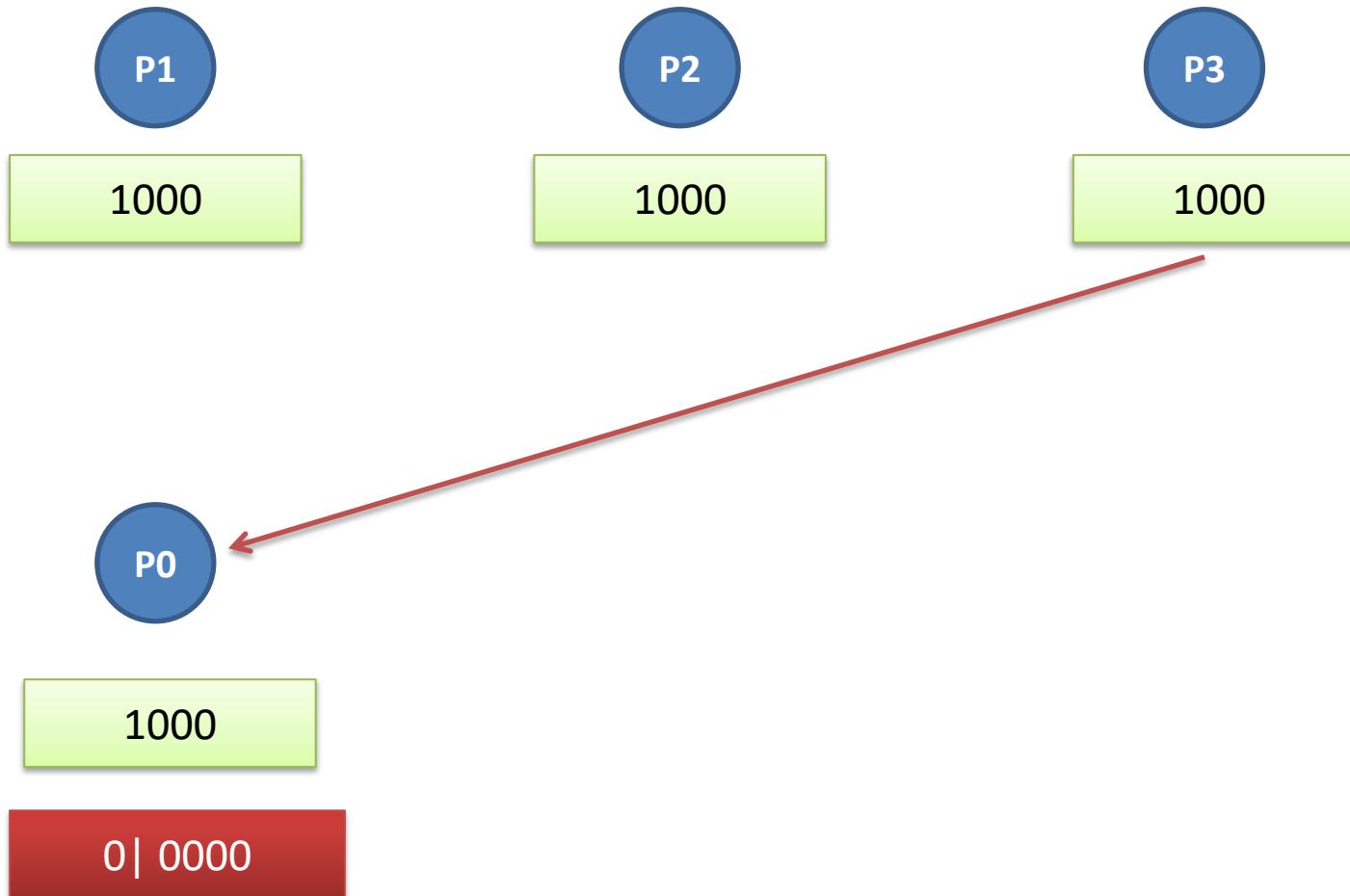
# Алгоритм широковещательный маркерный (Suzuki-Kasami)



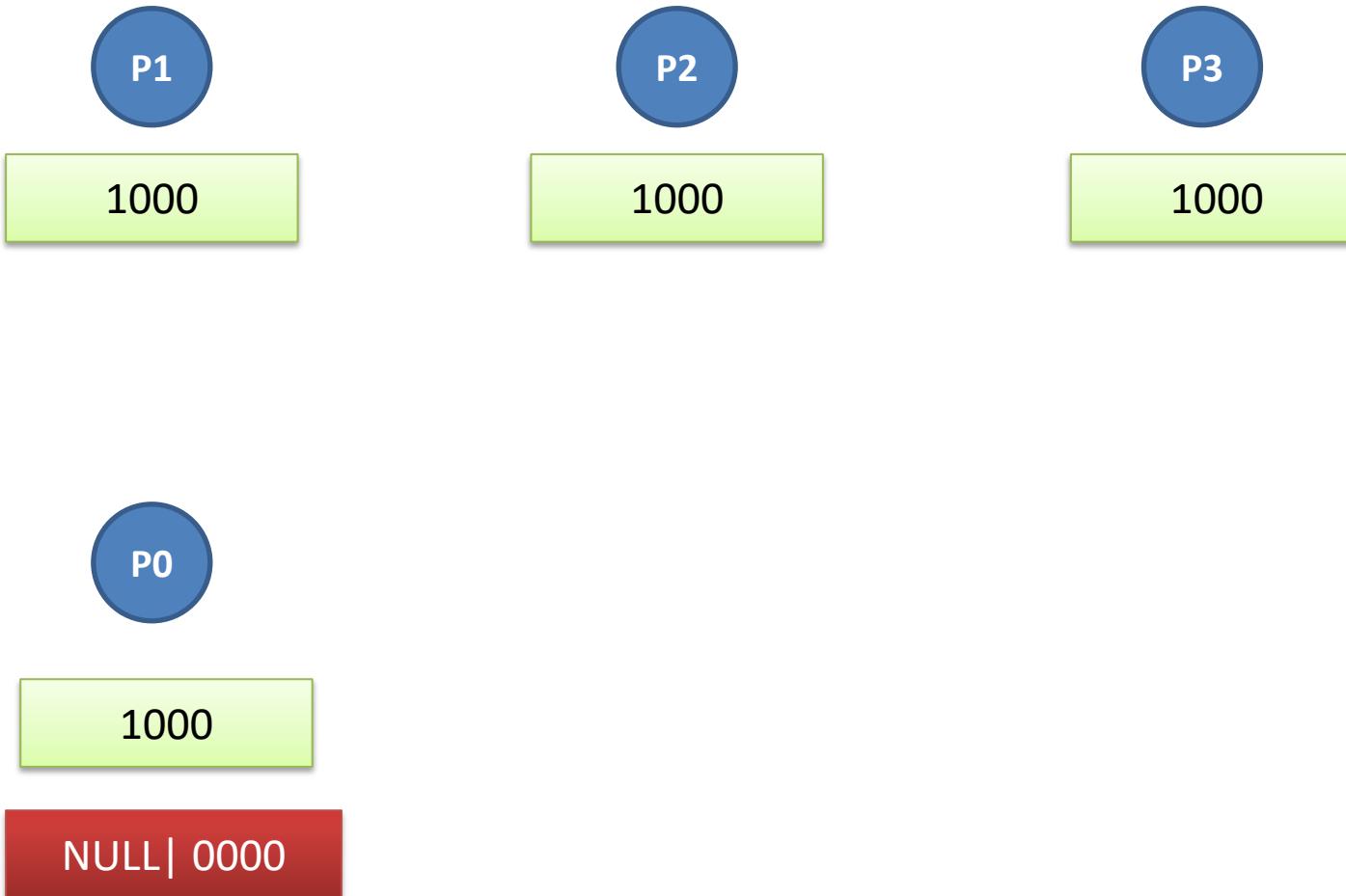
# Алгоритм широковещательный маркерный (Suzuki-Kasami)



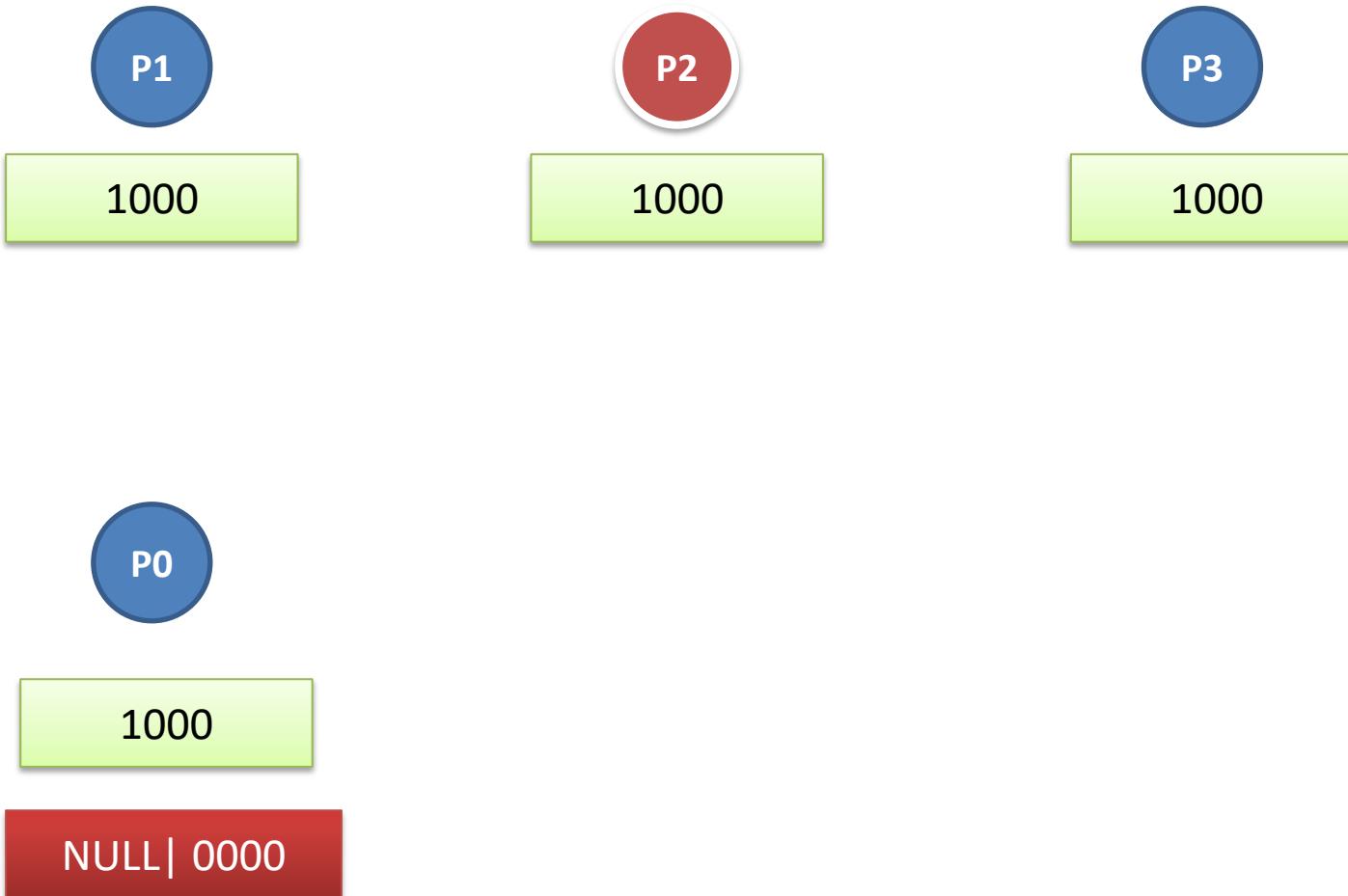
# Алгоритм широковещательный маркерный (Suzuki-Kasami)



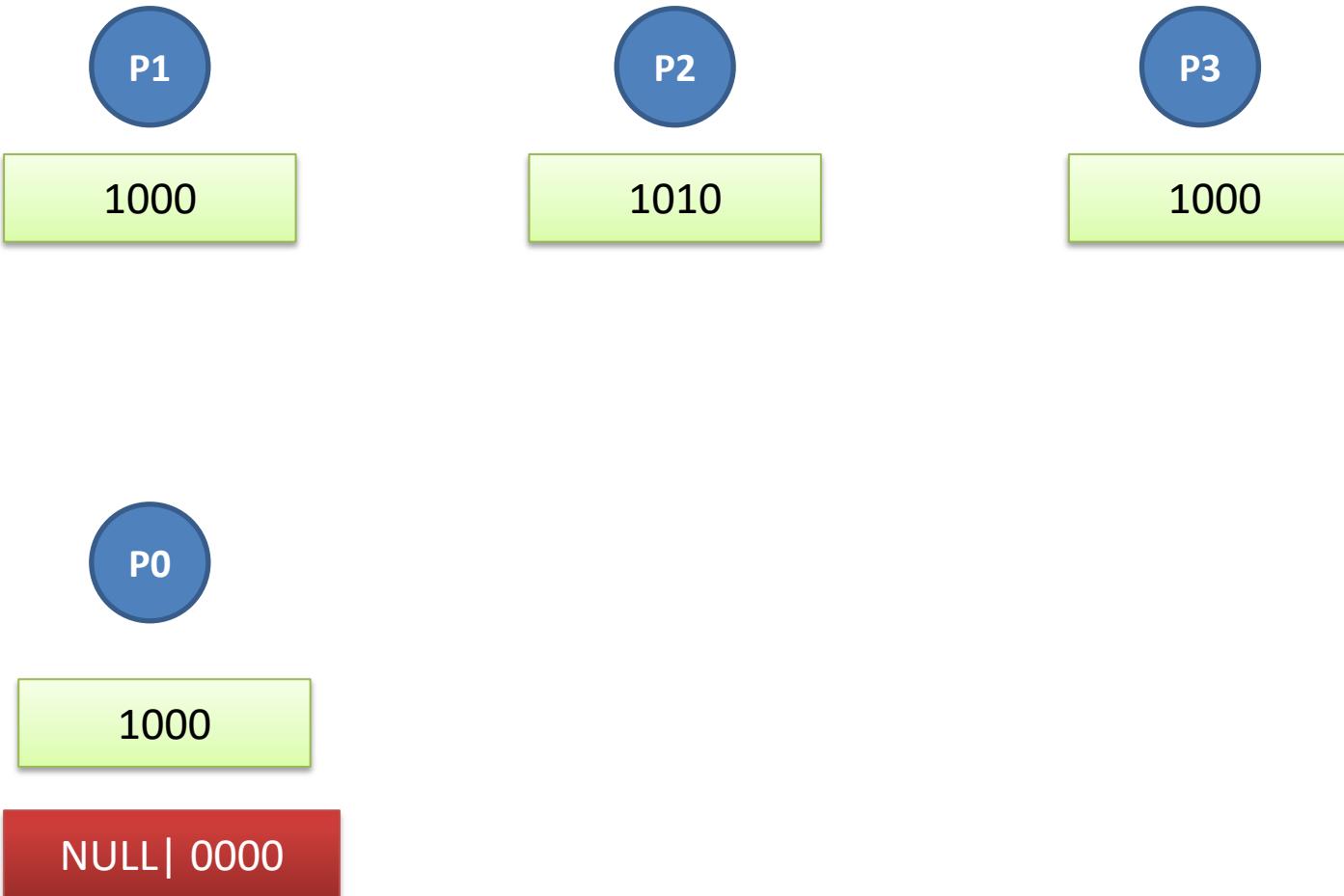
# Алгоритм широковещательный маркерный (Suzuki-Kasami)



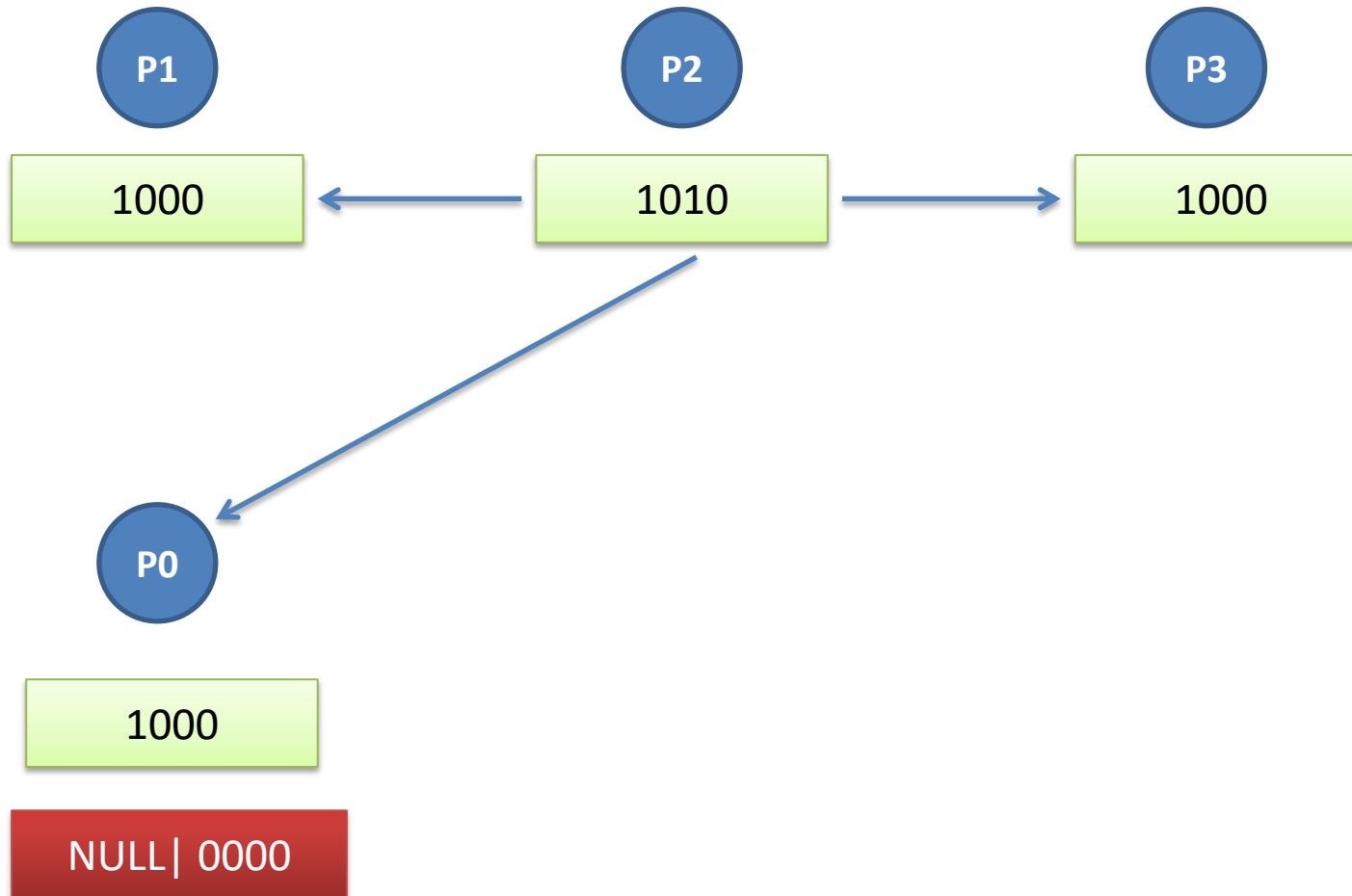
# Алгоритм широковещательный маркерный (Suzuki-Kasami)



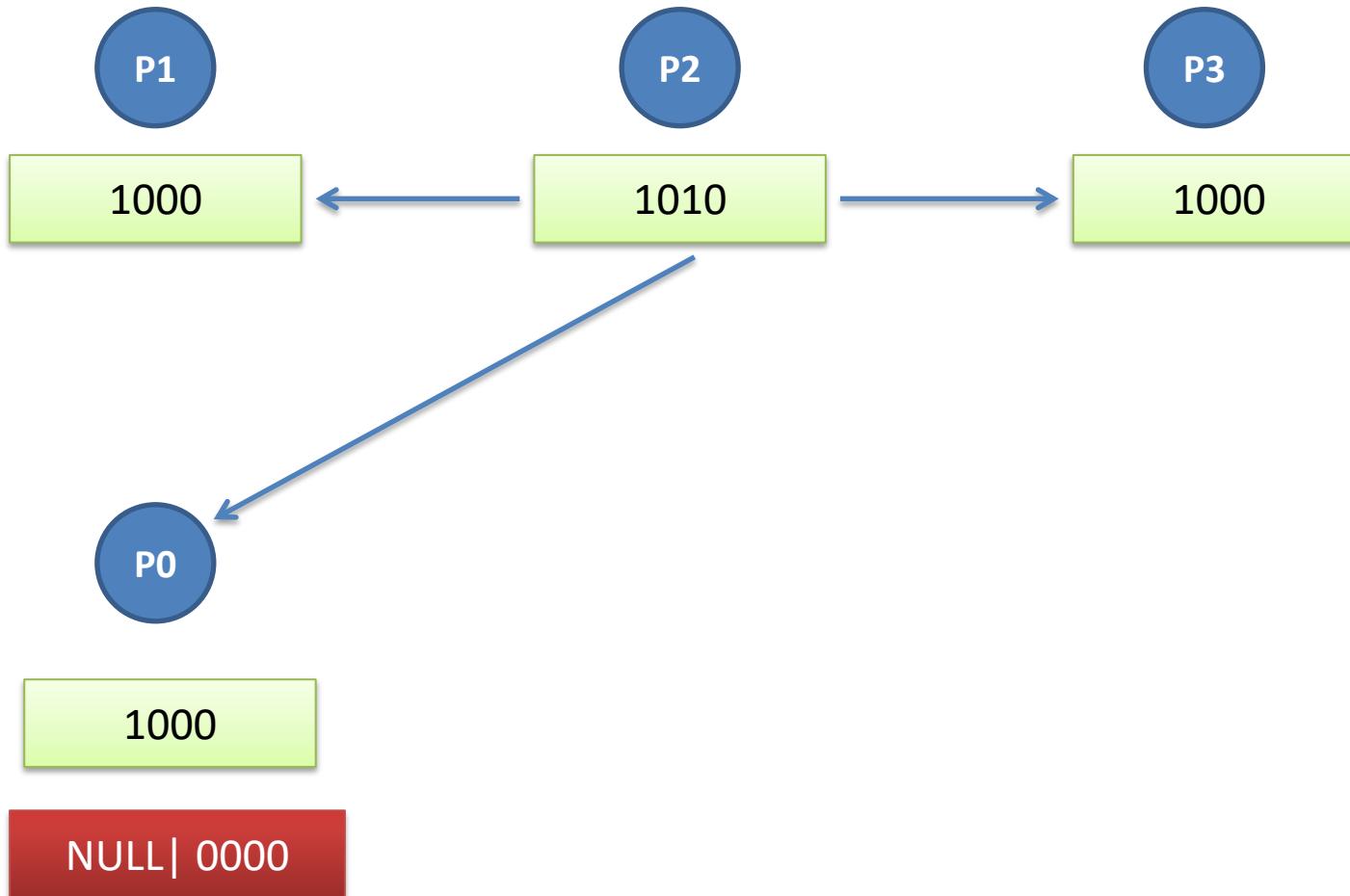
# Алгоритм широковещательный маркерный (Suzuki-Kasami)



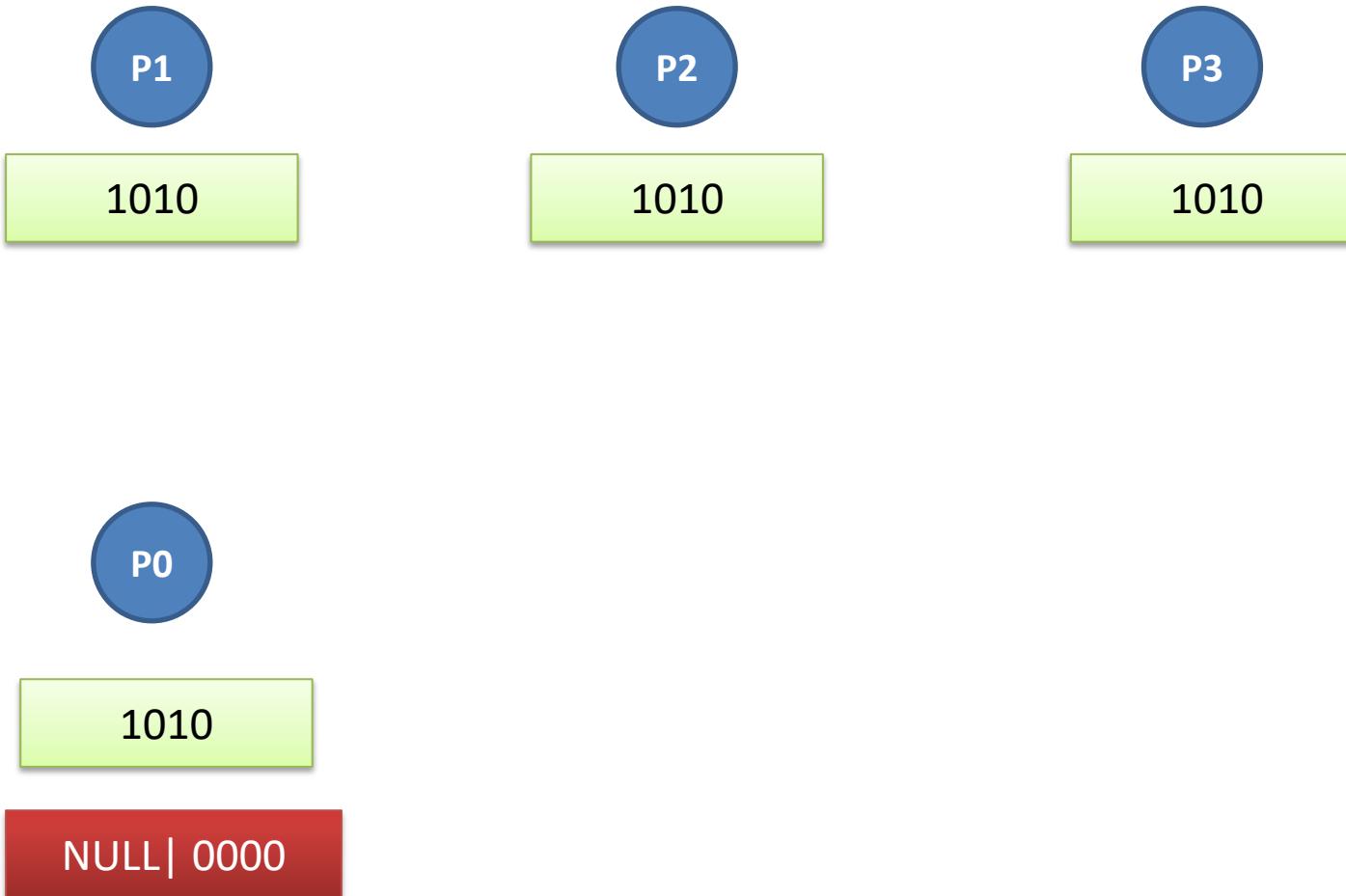
# Алгоритм широковещательный маркерный (Suzuki-Kasami)



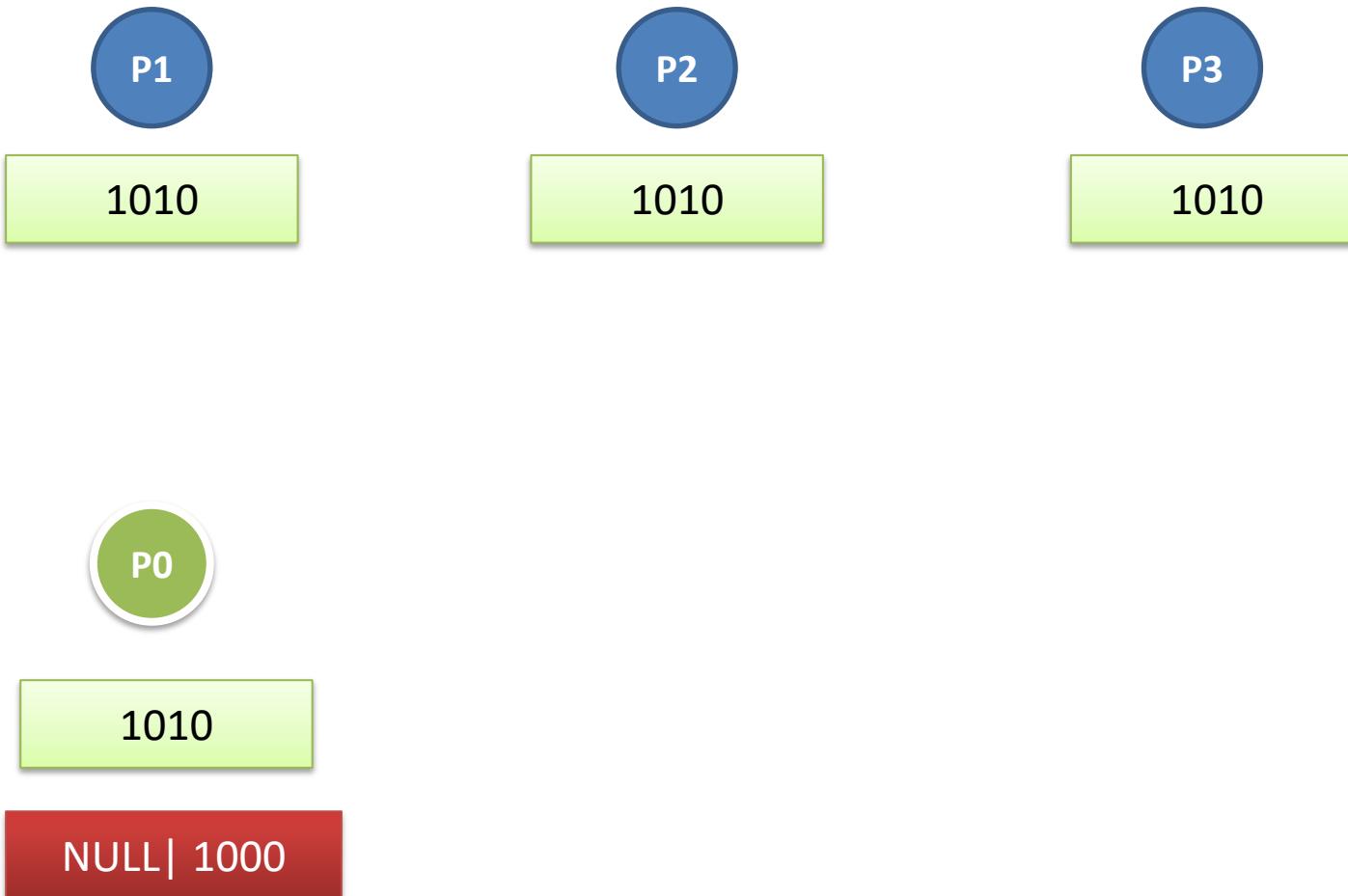
# Алгоритм широковещательный маркерный (Suzuki-Kasami)



# Алгоритм широковещательный маркерный (Suzuki-Kasami)



# Алгоритм широковещательный маркерный (Suzuki-Kasami)



# Алгоритм широковещательный маркерный (Suzuki-Kasami)

P1

1010

P2

1010

P3

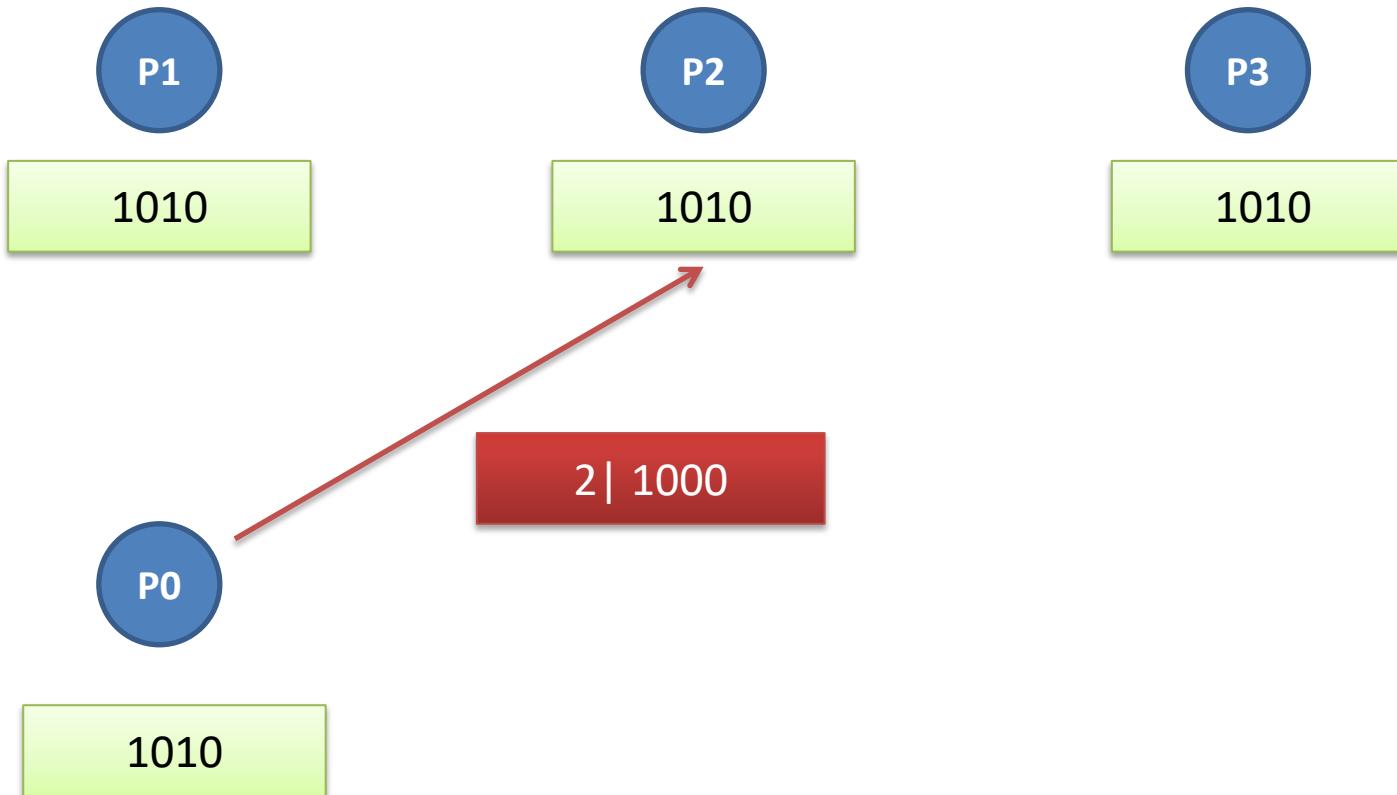
1010

P0

1010

2 | 1000

# Алгоритм широковещательный маркерный (Suzuki-Kasami)



# Алгоритм древовидный маркерный (Raymond)

Все процессы представлены в виде сбалансированного двоичного дерева. Каждый процесс имеет очередь запросов от себя и соседних процессов (1-го, 2-х или 3-х) и указатель в направлении владельца маркера.

## *Вход в критическую секцию*

- Если есть маркер, то процесс выполняет КС.
- Если нет маркера, то процесс:
  - 1) помещает свой запрос в очередь запросов;
  - 2) посылает сообщение «ЗАПРОС» в направлении владельца маркера и ждет сообщений.

# Алгоритм древовидный маркерный (Raymond)

## *Поведение процесса при приеме сообщений*

Процесс, не находящийся внутри КС должен реагировать на сообщения двух видов -«МАРКЕР» и «ЗАПРОС».

### **А) Пришло сообщение «МАРКЕР»:**

- М1. Взять 1-ый запрос из очереди и послать маркер его автору (концептуально, возможно себе).
- М2. Поменять значение указателя в сторону маркера.
- М3. Исключить запрос из очереди.
- М4. Если в очереди остались запросы, то послать сообщение «ЗАПРОС» в сторону маркера.

### **Б) Пришло сообщение «ЗАПРОС»:**

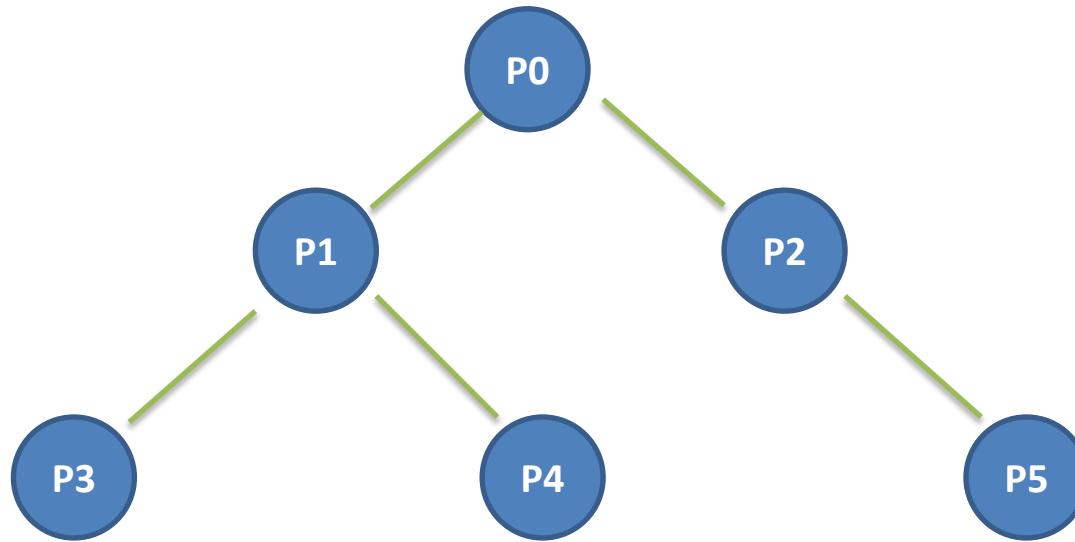
- Поместить запрос в очередь.
- Если нет маркера, то послать сообщение «ЗАПРОС» в сторону маркера, иначе (если есть маркер) - перейти на пункт М1.

# Алгоритм древовидный маркерный (Raymond)

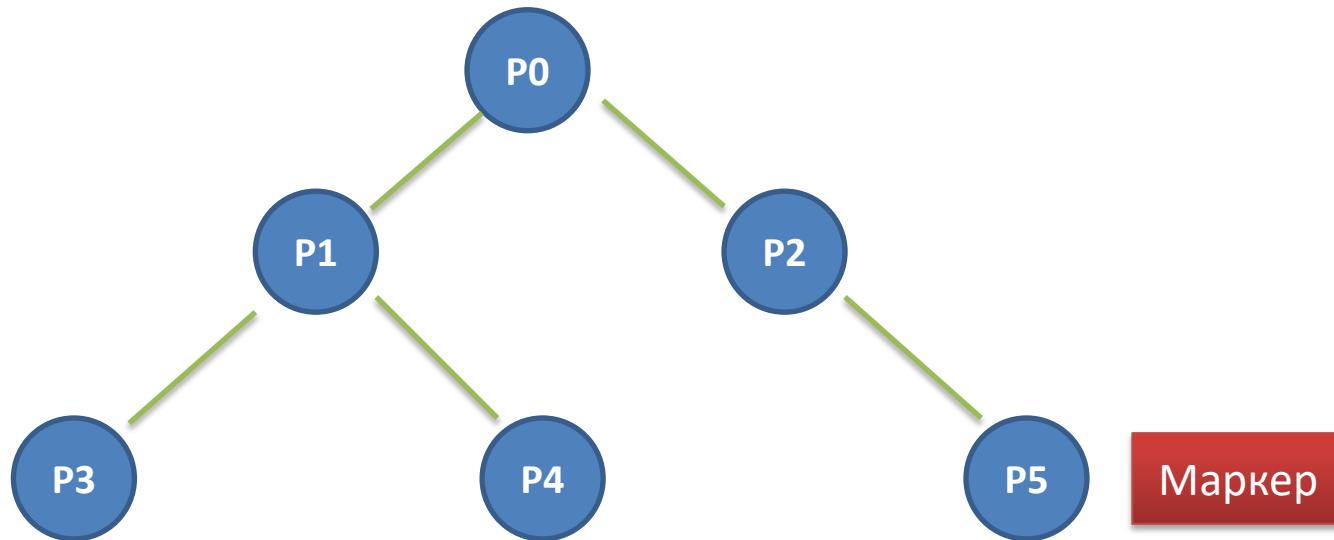
## *Выход из критической секции*

- Если очередь запросов пуста, то при выходе ничего не делается, иначе - перейти к пункту М1.

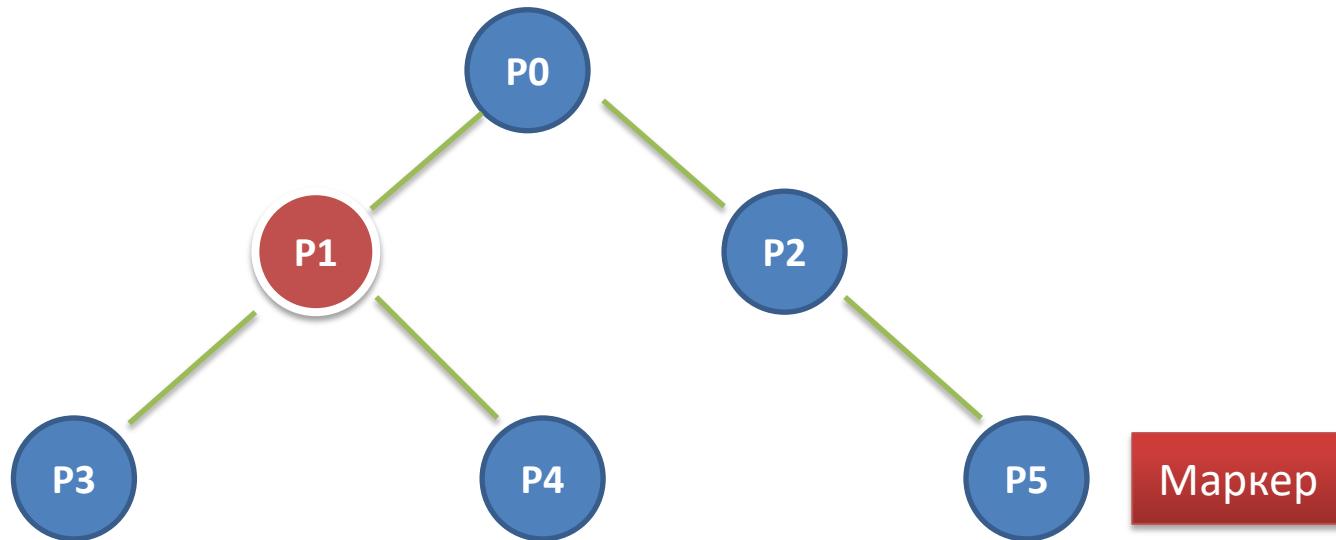
# Алгоритм древовидный маркерный (Raymond)



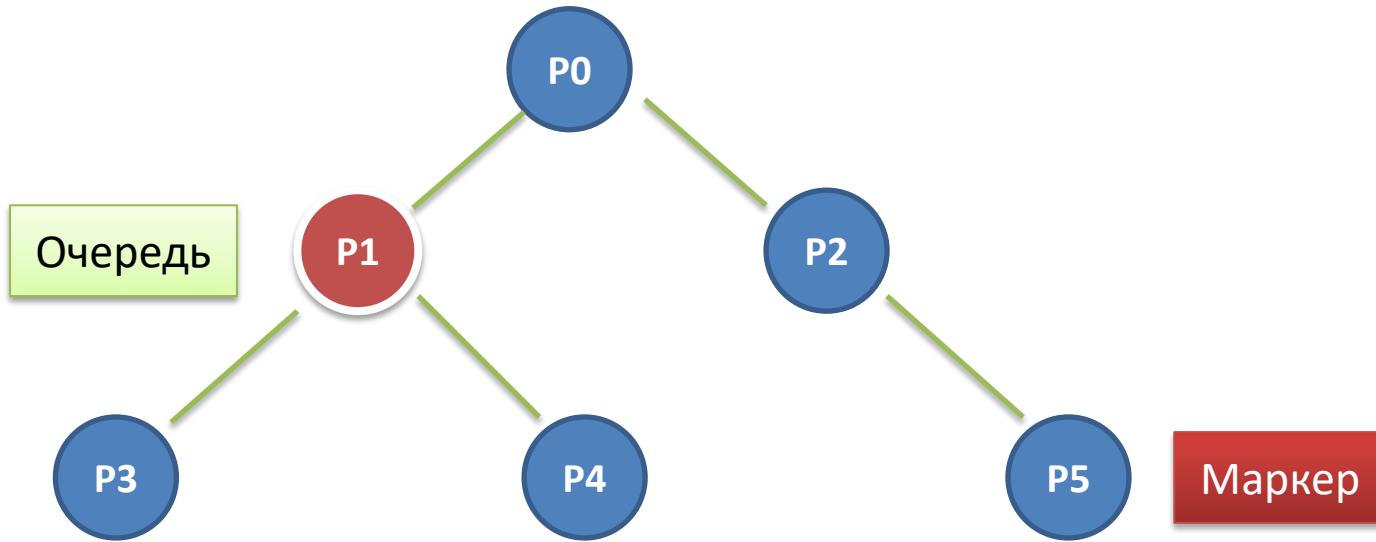
# Алгоритм древовидный маркерный (Raymond)



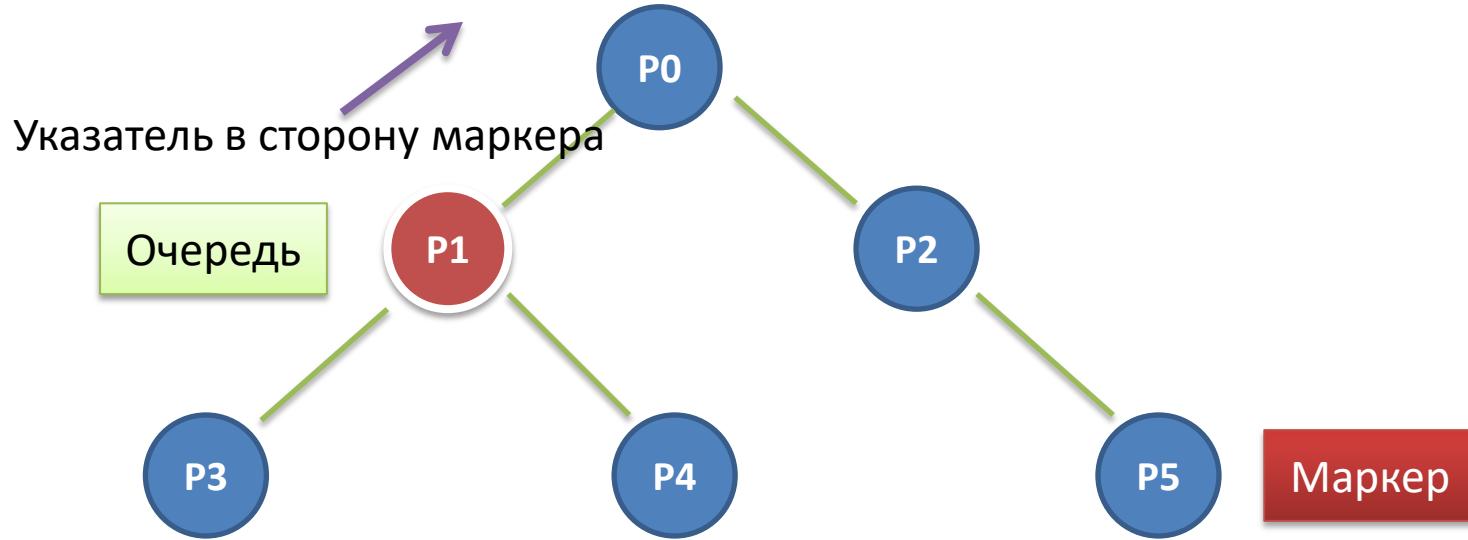
# Алгоритм древовидный маркерный (Raymond)



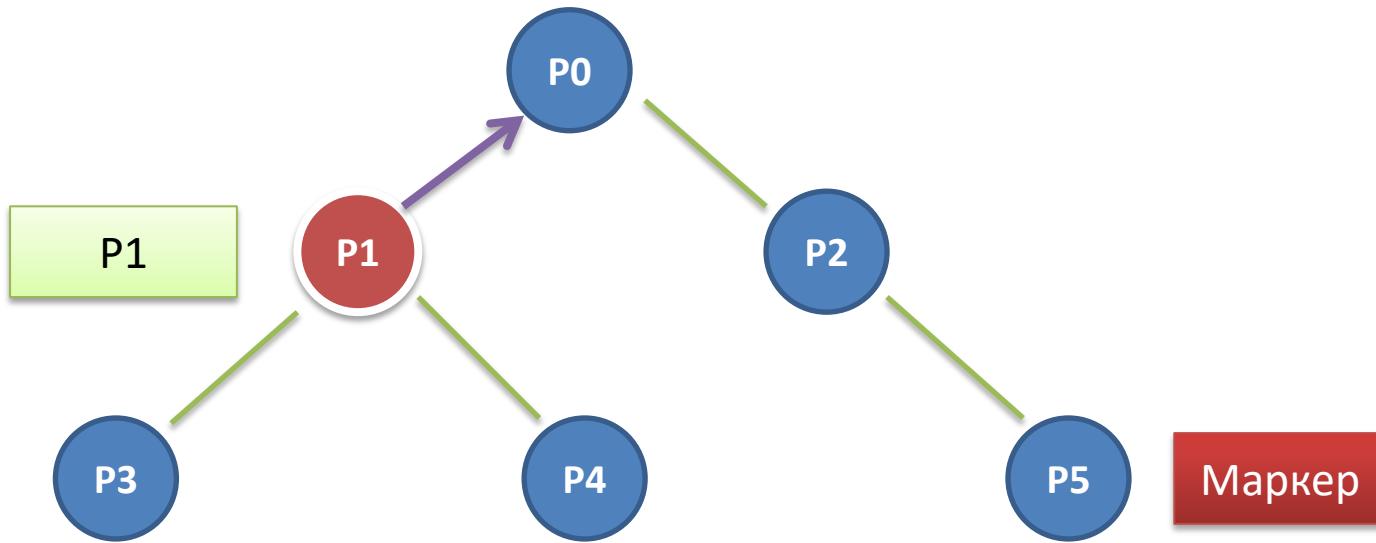
# Алгоритм древовидный маркерный (Raymond)



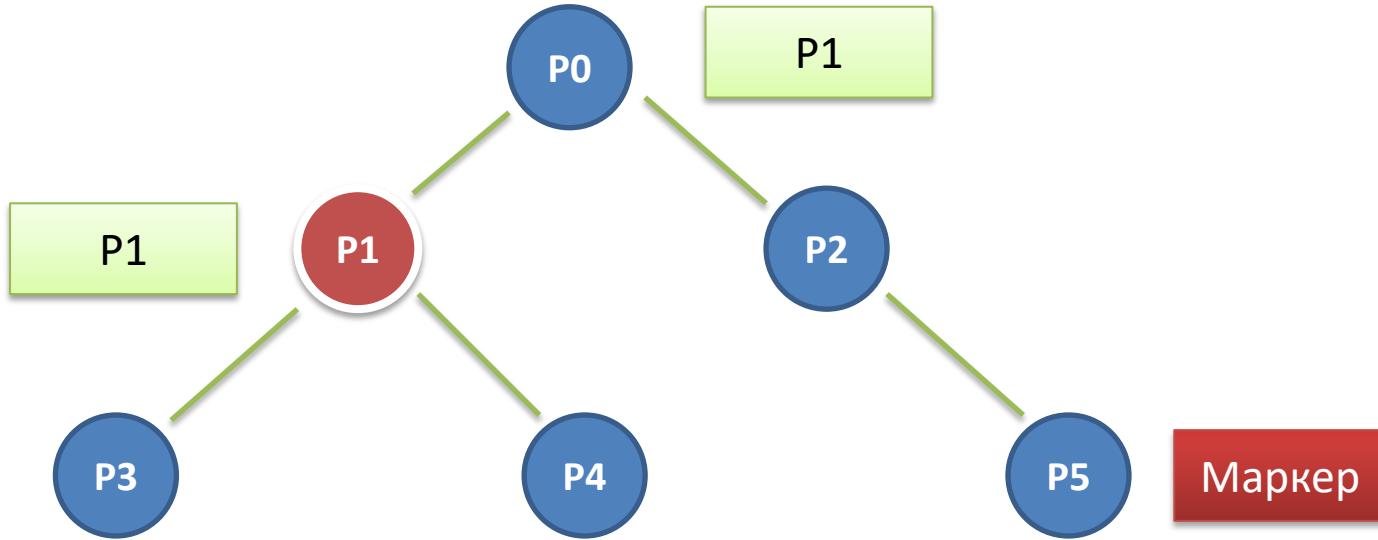
# Алгоритм древовидный маркерный (Raymond)



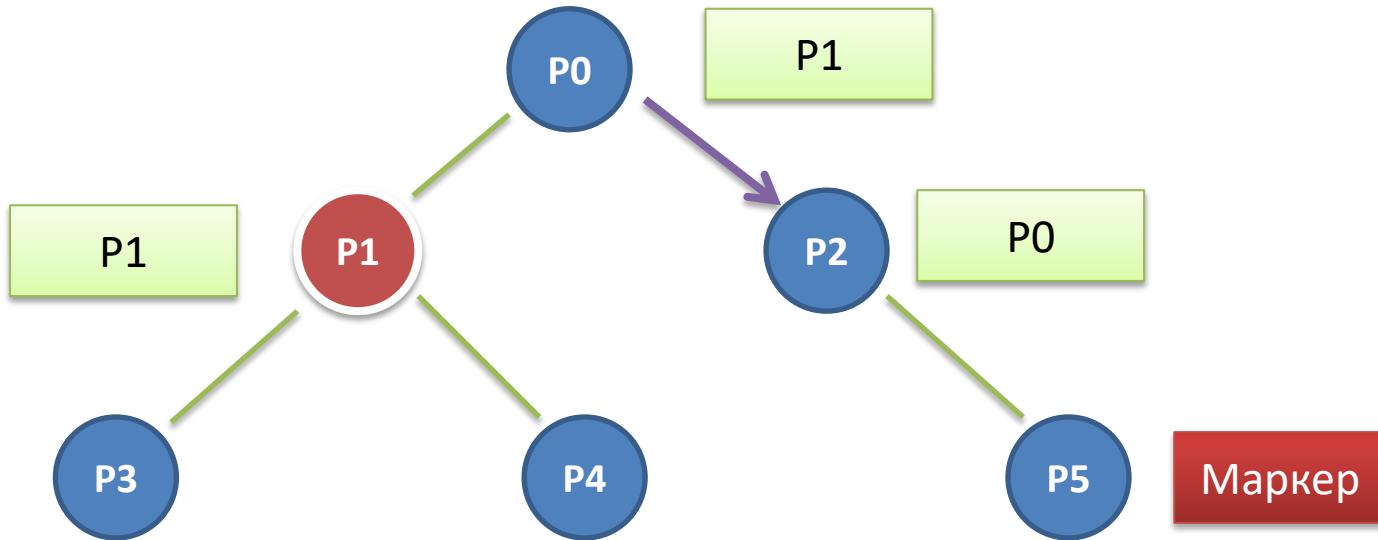
# Алгоритм древовидный маркерный (Raymond)



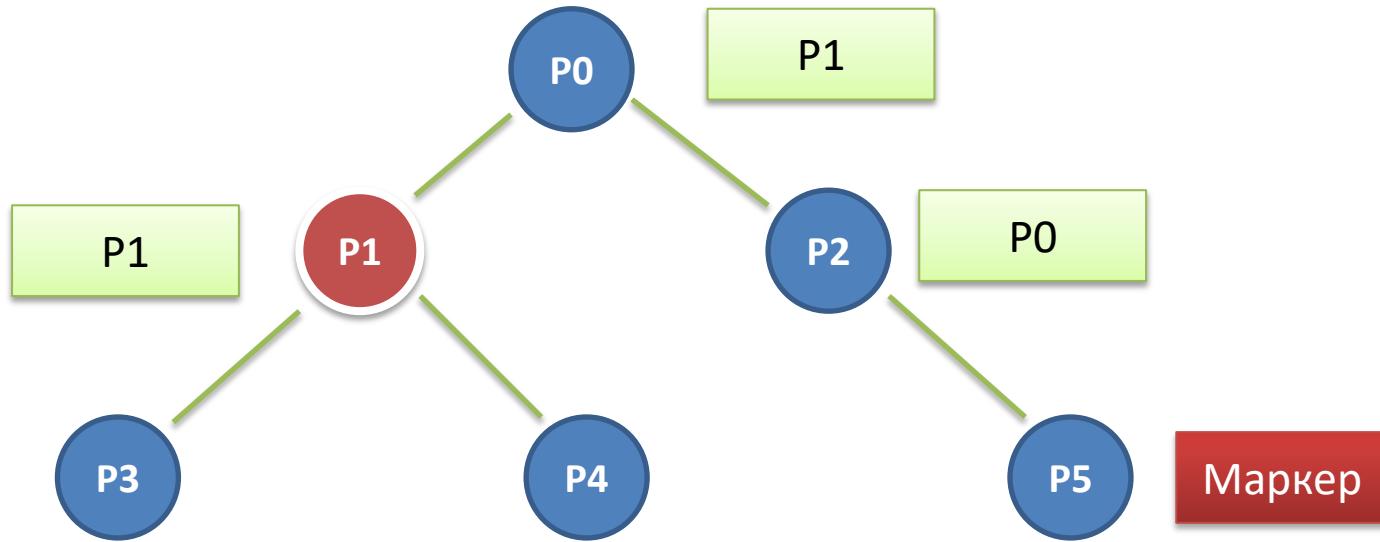
# Алгоритм древовидный маркерный (Raymond)



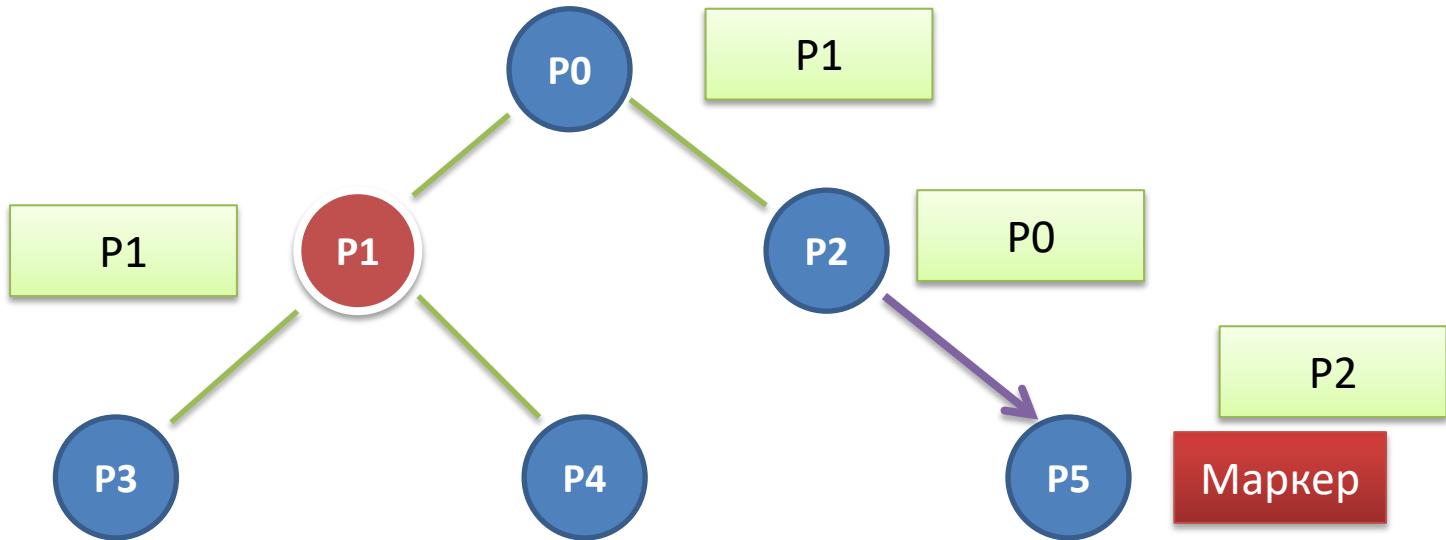
# Алгоритм древовидный маркерный (Raymond)



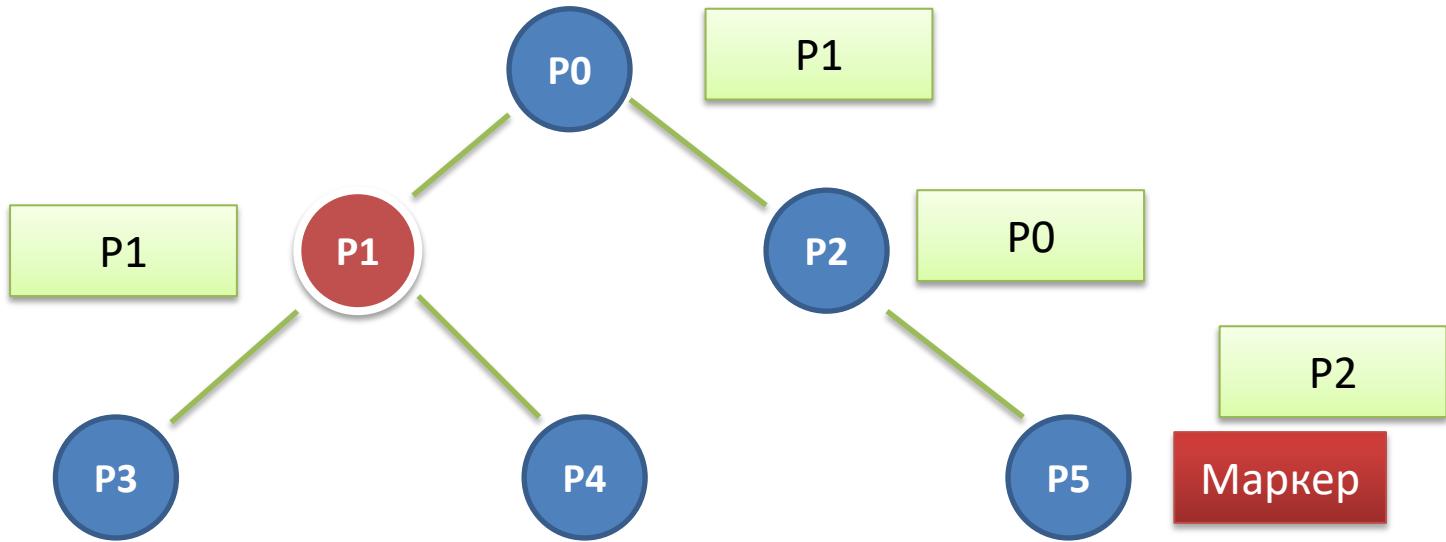
# Алгоритм древовидный маркерный (Raymond)



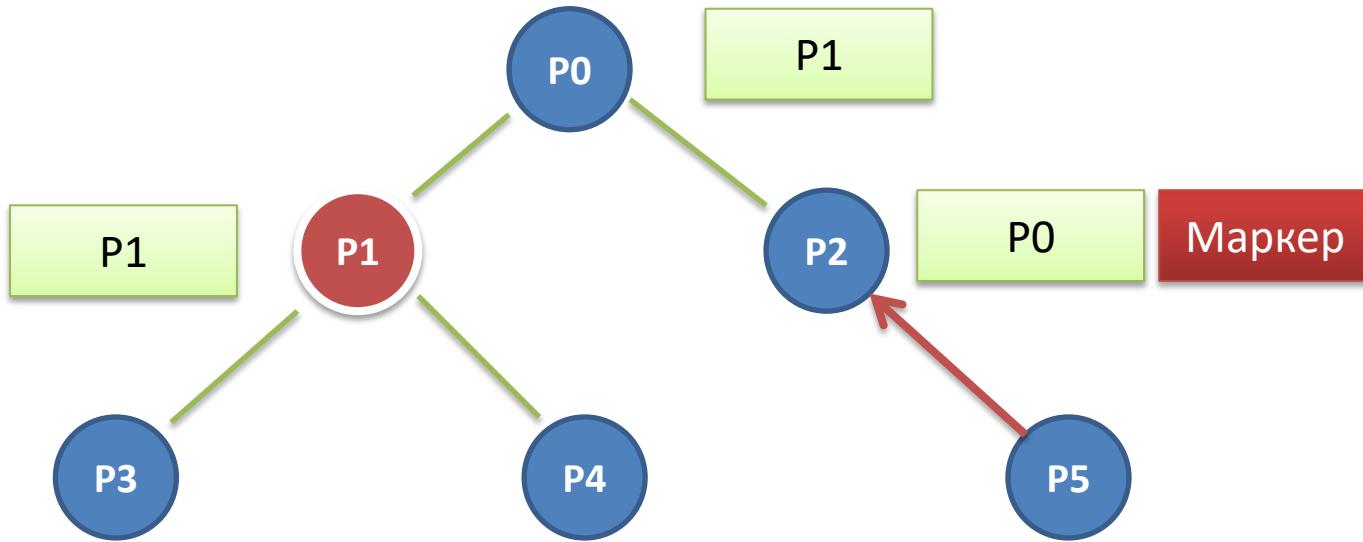
# Алгоритм древовидный маркерный (Raymond)



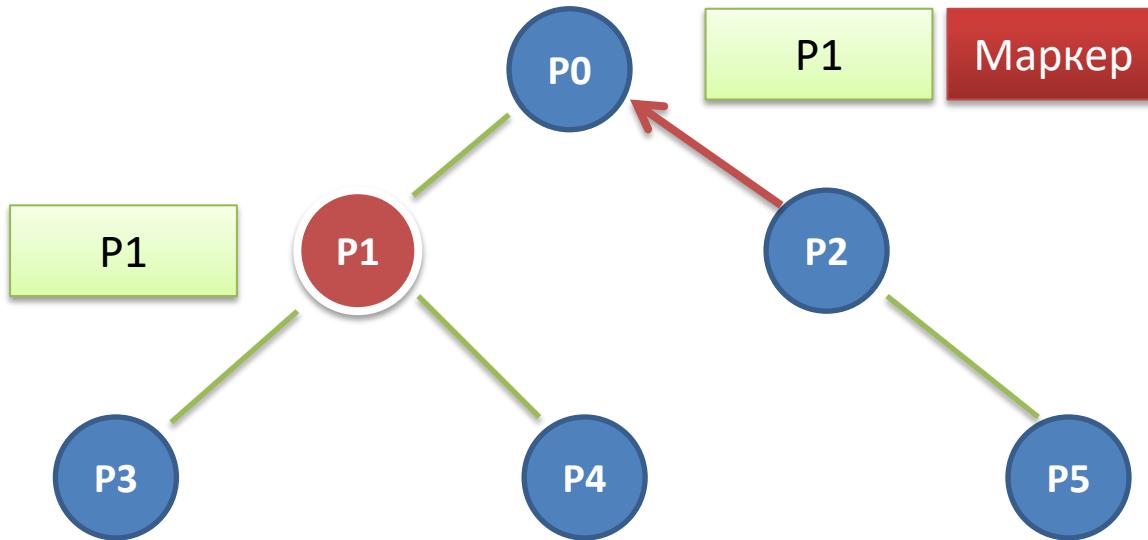
# Алгоритм древовидный маркерный (Raymond)



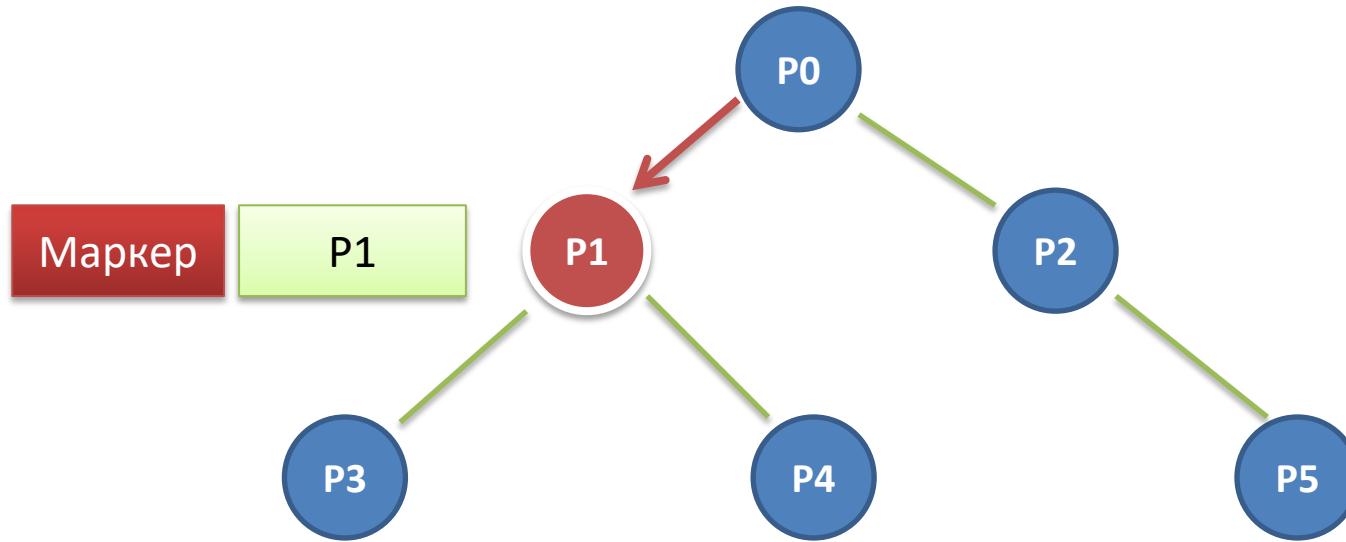
# Алгоритм древовидный маркерный (Raymond)



# Алгоритм древовидный маркерный (Raymond)



# Алгоритм древовидный маркерный (Raymond)



# Алгоритм древовидный маркерный (Raymond)

