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 памятями разных компьютеров аналогично тому, как они перемещаются между оперативной и внешней памятью одного компьютера.

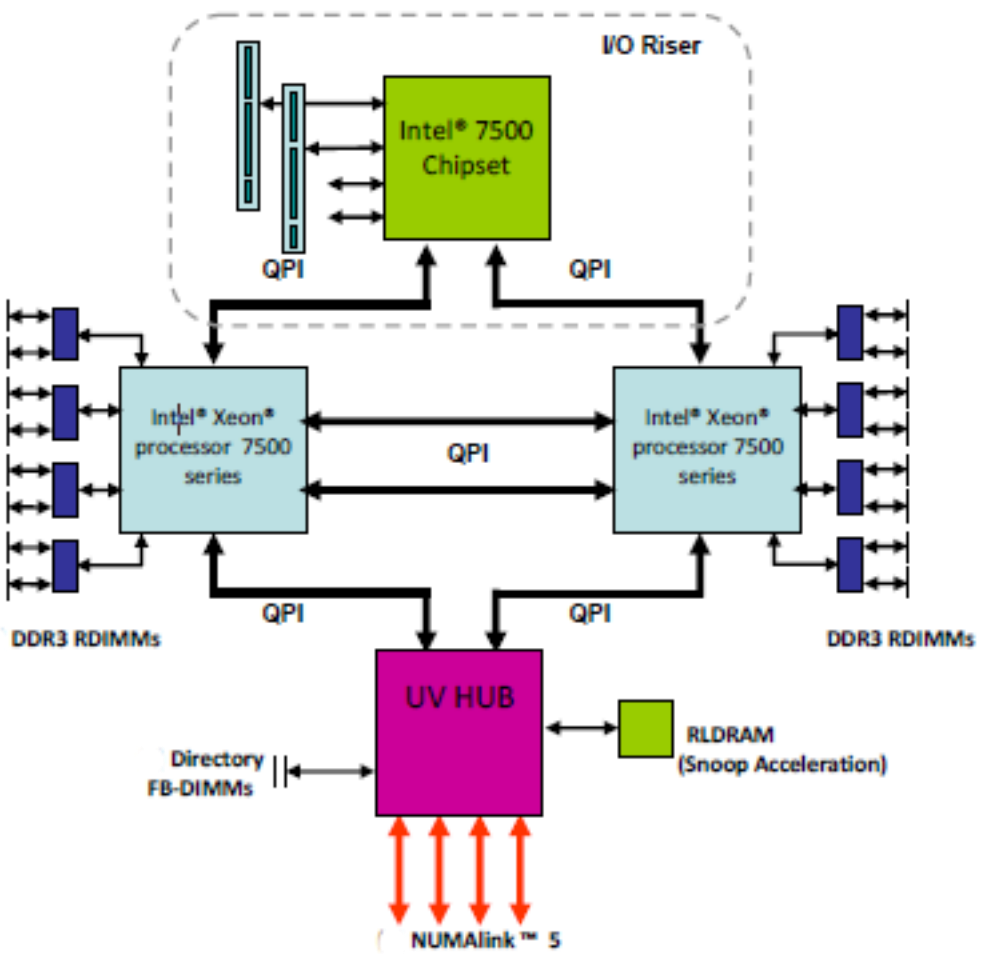
Достоинства DSM

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 беспредельно 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 (UltraVioloet) 2000

- 256 Intel® Xeon® processor E5-4600 product family 2.4GHz-3.3GHz - 2048 Cores (4096 Threads)
- 64 TB 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 посылает 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 насильно 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 посылать свои модификации этому процессу. Он будет присваивать каждой модификации очередной номер и рассылать 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.

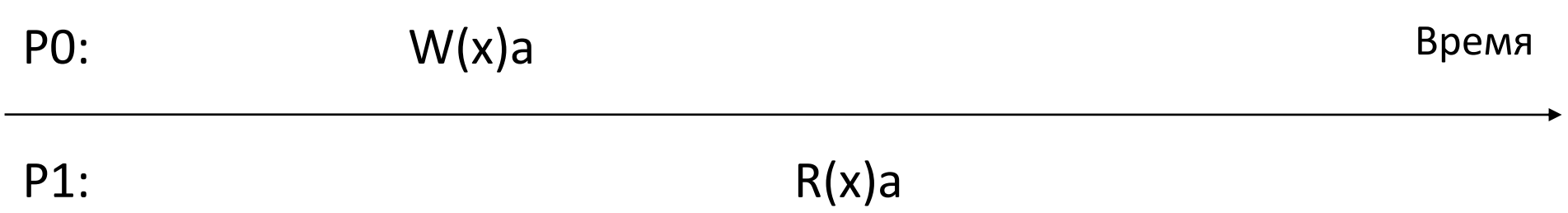
Модели консистентности

- 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.

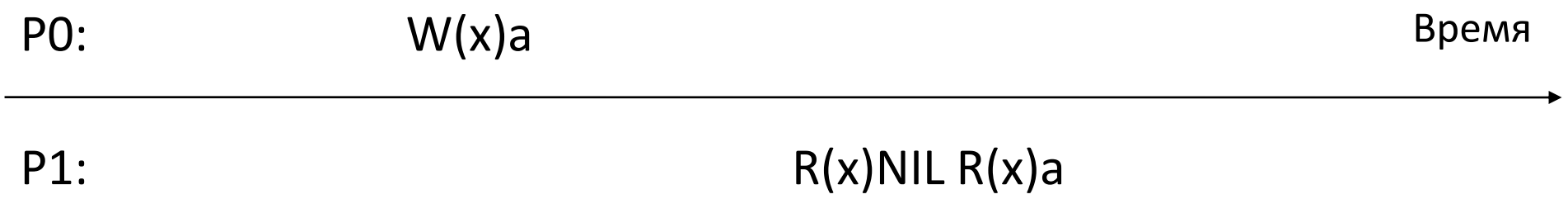
Строгая консистентность

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

б) не удовлетворяет последовательной консистентности

P1	**W(x)a**				
P2		**W(x)b**			
P3			**R(x)b**		**R(x)a**
P4				**R(x)a**	**R(x)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; Print (y,z);	y=1; Print(x,z);	z=1; Print (x,y);

x=1;	x=1;	y=1;	y=1;
Print (y,z);	y=1;	z=1;	x=1;
y=1;	Print(x,z);	Print (x,y);	z=1;
Print(x,z);	Print (y,z);	Print(x,z);	Print(x,z);
z=1;	z=1;	x=1;	Print (y,z);
Print (x,y);	Print (x,y);	Print (y,z);	Print (x,y);
001011	101011	0101111	111111

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 рассылает 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 рассылается 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	Процесс P2
.....	.....
a = 1;	b = 1;
if (b==0) kill (P2);	if (a==0) kill (P1);
.....	.....

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

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 рассылает 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 рассылаться 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 последователь-ность 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

- Следующие правила определяют требования к модели консистентности по выходу:
- До выполнения обращения к общей переменной, должны быть полностью выполнены все предыдущие захваты синхронизационных переменных данным процессором.
- Перед освобождением синхронизационной переменной должны быть закончены все операции чтения/записи, выполнявшиеся процессором прежде.
- Реализация операций захвата и освобождения синхронизационной переменной должны удовлетворять требованиям процессорной консистентности (последовательная консистентность не требуется).

P1	**Acq(L)**	**W(x)a**	**W(x)b**	**Rel(L)**				
P2					**Acq(L)**	**R(x)b**	**Rel(L)**	
P3								**R(x)a**

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 посылая 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

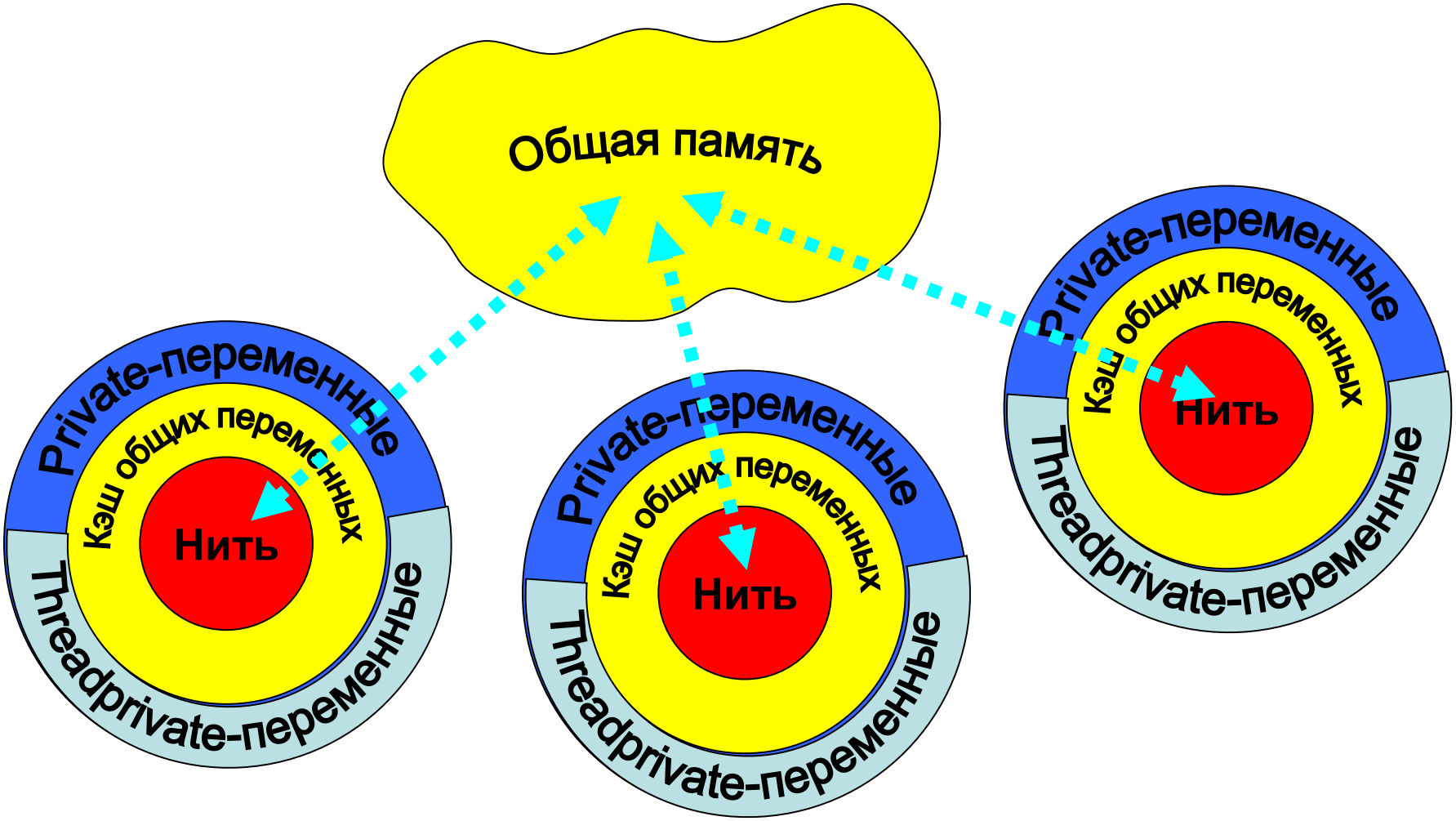
• Процесс не может захватить синхронизационную переменную до того, пока не обновлены все переменные этого процесса, охраняемые захватываемой синхронизационной переменной;
• Процесс не может захватить синхронизационную переменную в монопольном режиме (для модификации охраняемых данных), пока другой процесс, 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

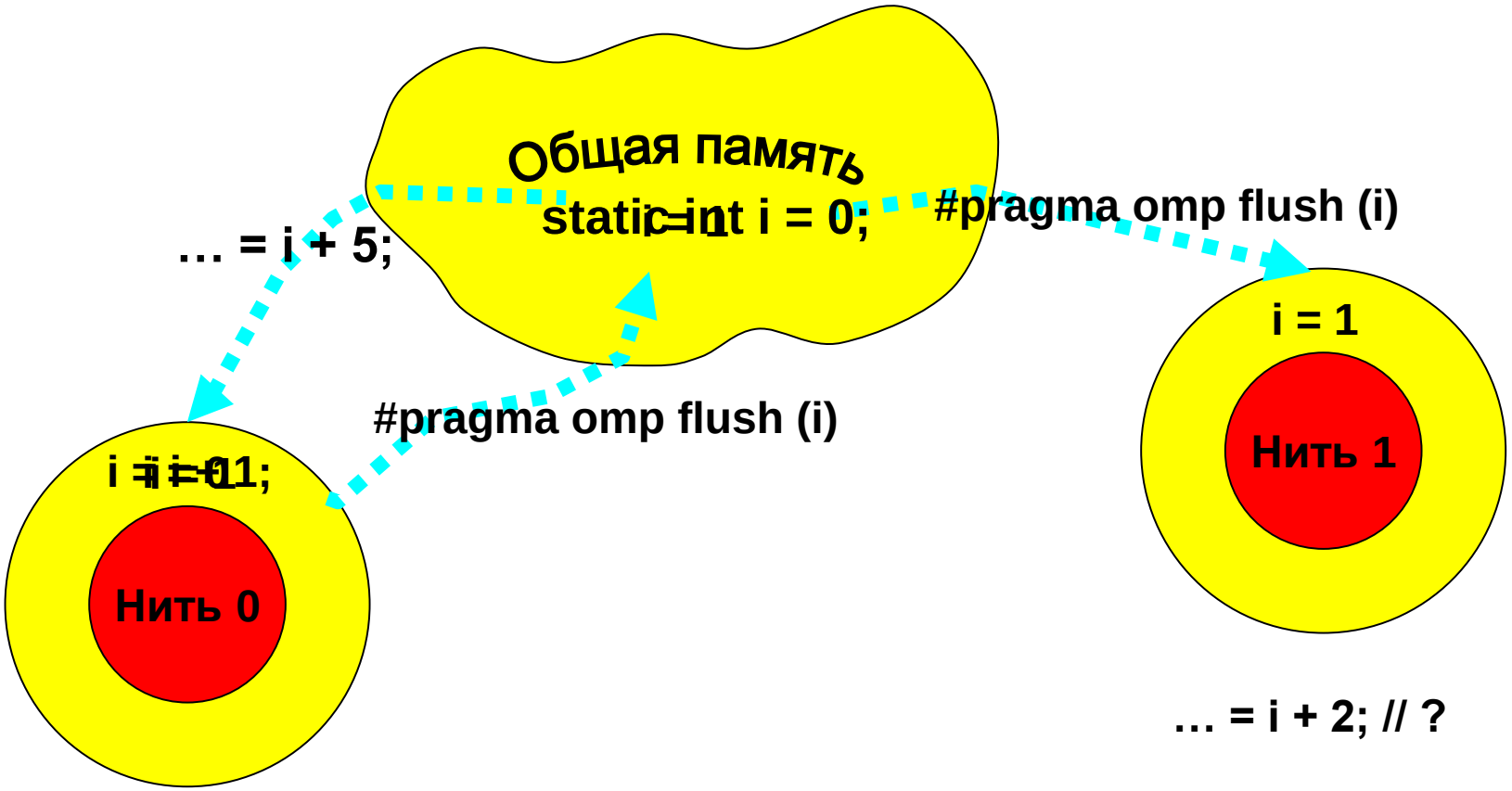
Консистентность	Описание
Строгая	Упорядочение всех доступов к разделяемым данным по абсолютному времени
Последовательная	Все процессы видят все записи разделяемых данных в одном и том же порядке
Причинная	Все процессы видят все причинно-связанные записи данных в одном и том же порядке
Процессорная	PRAM-консистентность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

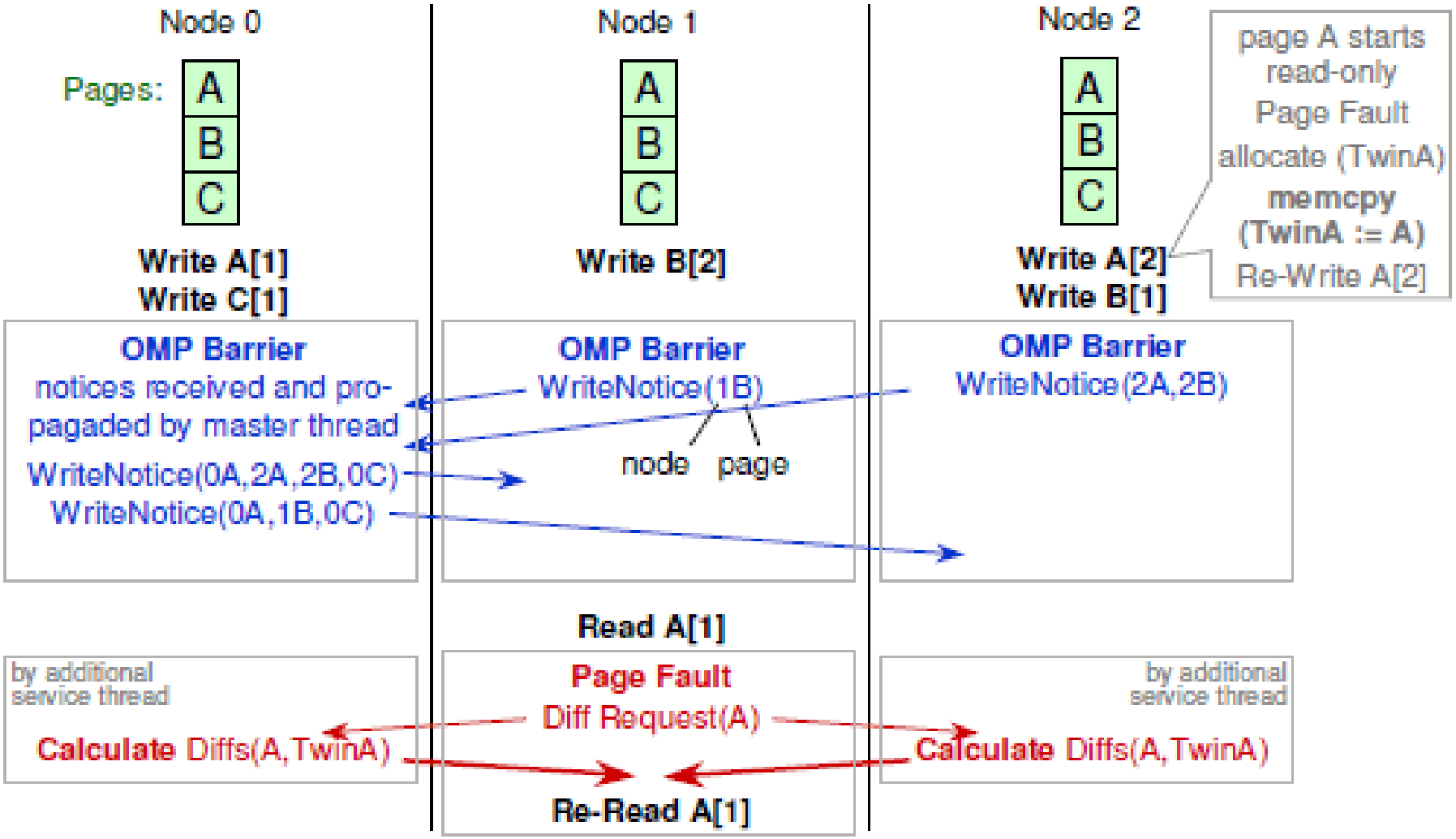
!dir$ omp sharable (variable *[, variable …]*) - для Фортрана

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;
}
```