Интернет Университет Суперкомпьютерных технологий

Лекция 2: OpenMP - модель параллелизма по управлению

Учебный курс

Параллельное программирование с OpenMP

Бахтин В.А., кандидат физ.-мат. наук,
заведующий сектором,
Институт прикладной математики им. М.В.Келдыша РАН

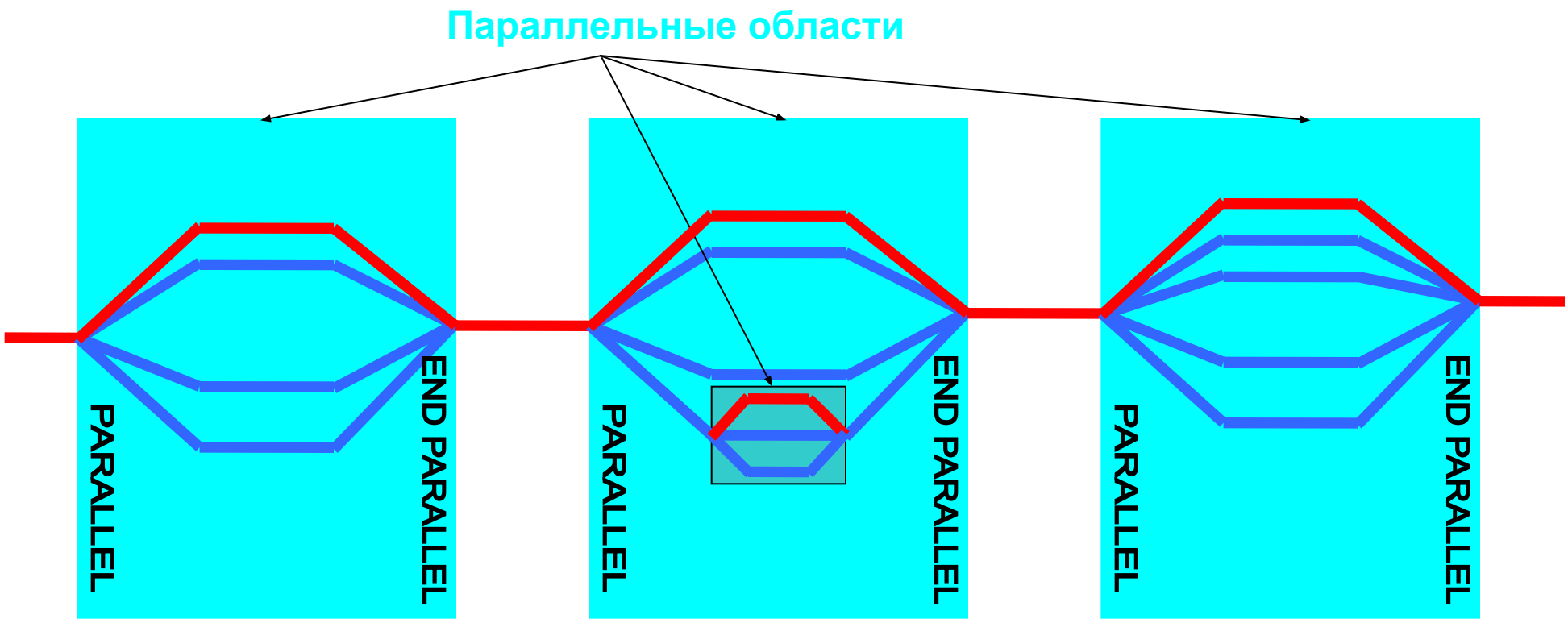
Содержание

- Выполнение OpenMP-программы (Fork and Join Model).
- Модель памяти. Понятие консистентности памяти.
- Консистентность памяти в OpenMP (weak ordering).
- Классы переменных (клаузы SHARED, PRIVATE; директива THREADPRIVATE).

Выполнение OpenMP-программы

Fork-Join параллелизм:

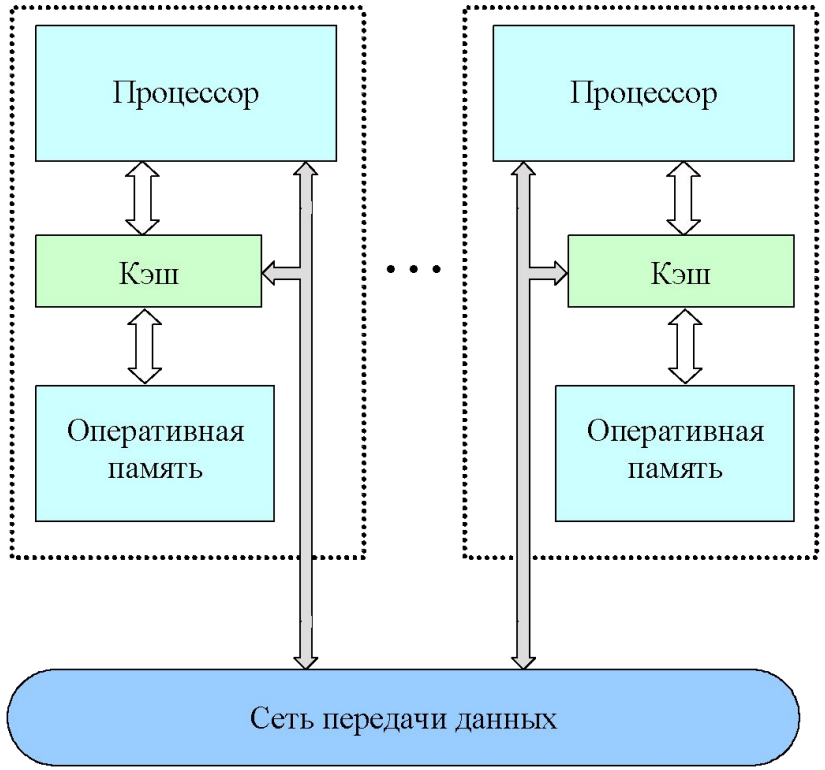
- Главная (master) нить порождает группу (team) нитей по мере небходимости.
- Параллелизм добавляется инкрементально.



Когерентность и консистентность памяти

P0: W(x)a R(y)b Время

P1: W(y)b R(y)b R(x)a W(x)b



Сеть передачи данных

Сеть передачи данных	**Производитель**	**MPI latency, в микросекундах**	**Bandwidth per link** (unidirectional, MB/s)
NUMAlink 4 (Altix)	SGI	1	3400
RapidArray (XD1)	Cray	1.8	2000
QsNet II	Quadrics	2	900
Infiniband	Voltaire	3.5	830
High Performance Switch	IBM	5	1000
Myrinet XP2	Myricom	5.7	495
SP Switch 2	IBM	18	500
Ethernet	Various	30	100

http://www.sgi.com/products/servers/altix/numalink.htm

Модели консистентности памяти

- **Модель консистентности** представляет собой некоторый договор между программами и памятью, в котором указывается, что при соблюдении программами определенных правил работа памяти будет корректной, если же требования к программе будут нарушены, то память не гарантирует правильность выполнения операций чтения/записи.

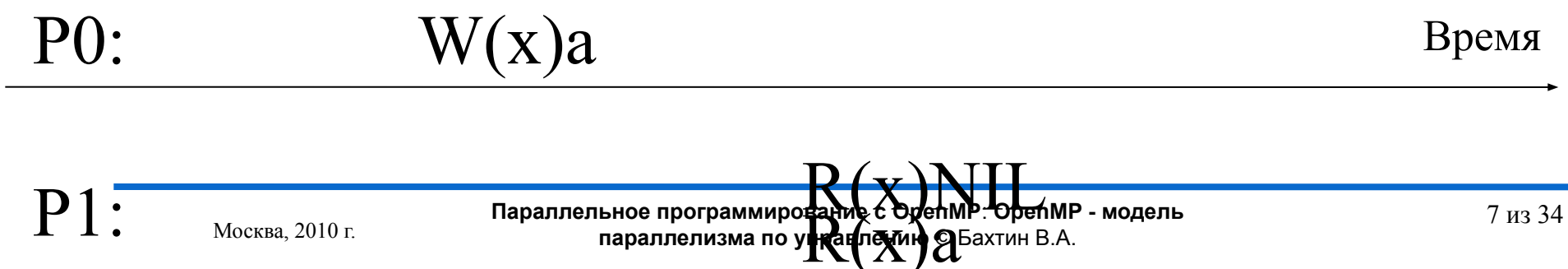
- Далее рассматриваются основные модели консистентности.

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, причем операции каждого отдельного процесса выполнялись бы в порядке, определяемой его программой.
- 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 знают последовательность всех 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);

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.

PRAM (Pipelined RAM) и процессорная консистентность

PRAM: 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.

Процессорная: PRAM + 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.

Слабая консистентность (weak consistency)

- 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.

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 (weak consistency)

- 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 (weak consistency)

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.

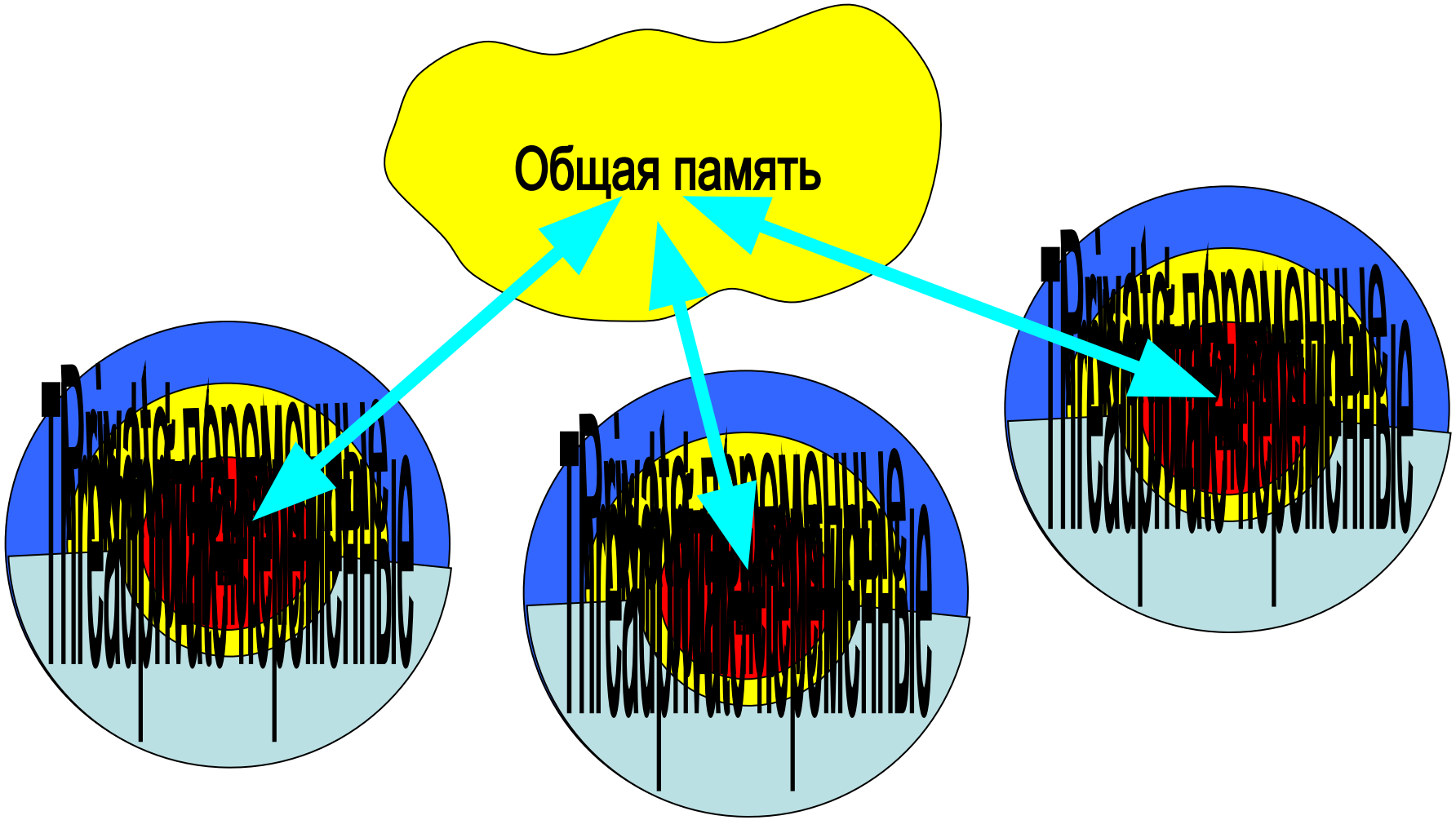
Консистентность по выходу

Следующие правила определяют требования к модели консистентности по выходу:

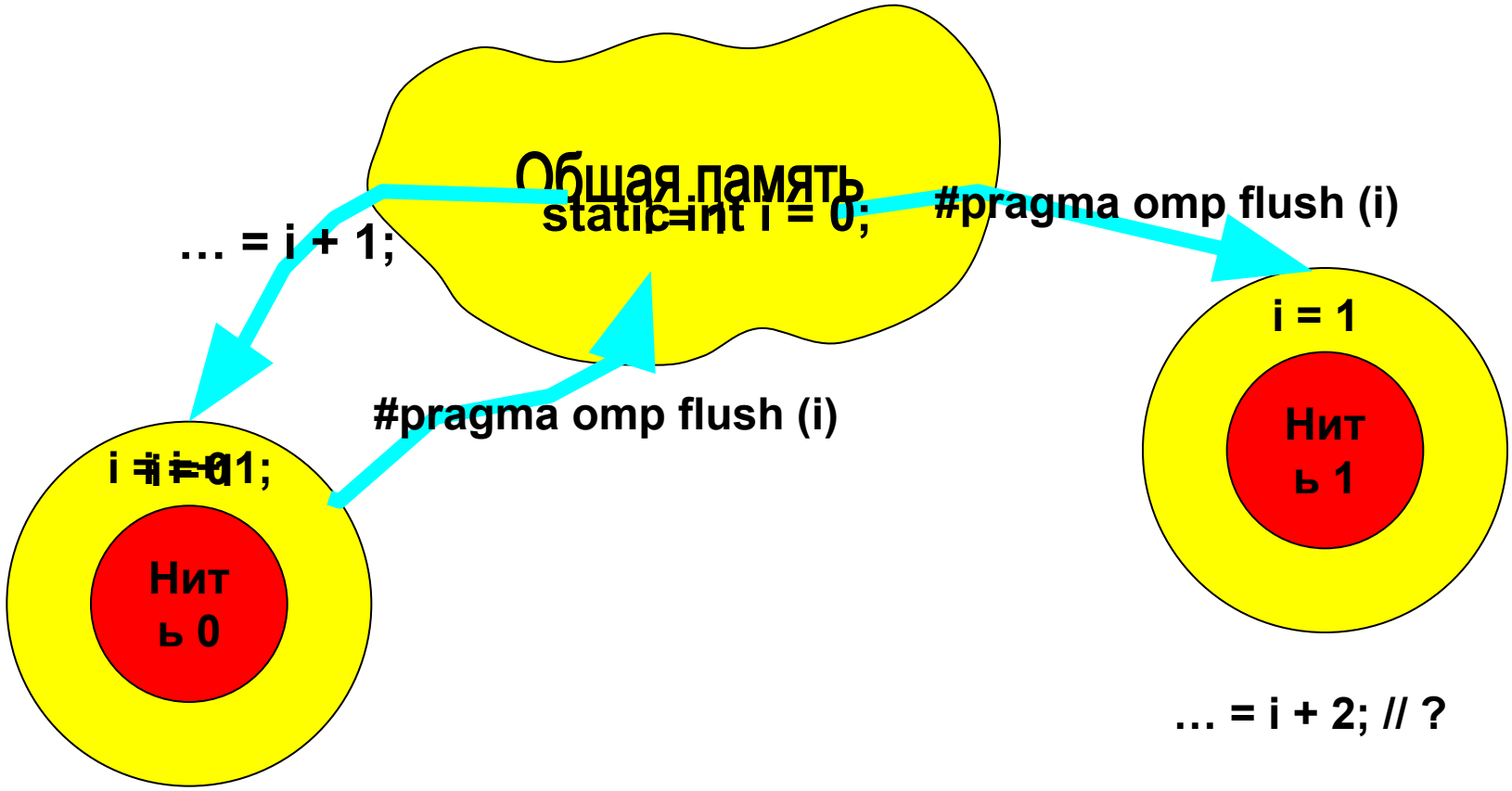
- До выполнения обращения к общей переменной, должны быть полностью выполнены все предыдущие захваты синхронизационных переменных данным процессором.
- Перед освобождением синхронизационной переменной должны быть закончены все операции чтения/записи, выполнявшиеся процессором прежде.
- Реализация операций 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**

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)

Директива flush:

#pragma omp flush *[(list)]* - для Си

!$omp flush *[(list)]* - для Фортран

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).

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**.

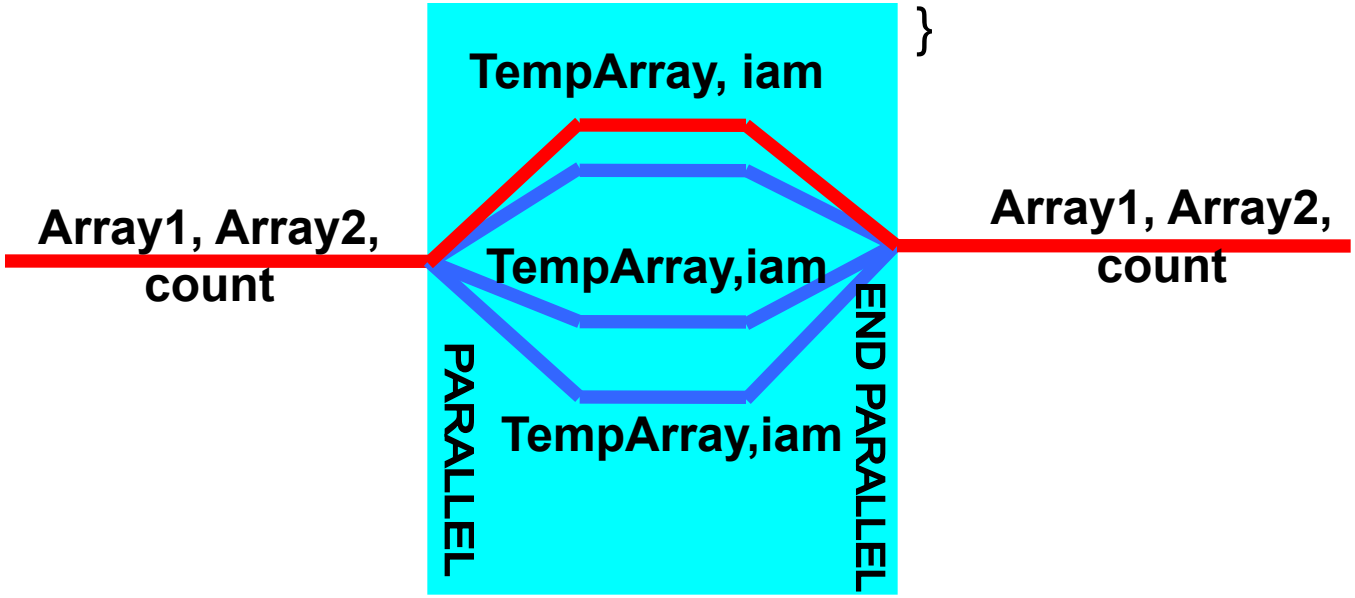
Классы переменных

- В модели программирования с разделяемой памятью:
  - Большинство переменных по умолчанию считаются SHARED
- Глобальные переменные совместно используются всеми нитями (shared)
  - Фортран: COMMON блоки, SAVE переменные, MODULE переменные
  - Си: file scope, static
  - Динамически выделяемая память (ALLOCATE, malloc, new)
- Но не все переменные являются разделяемыми ...
  - Стековые переменные в подпрограммах (функциях), вызываемых из параллельного региона, являются PRIVATE.
  - Переменные, объявленные внутри блока операторов параллельного региона являются приватными.
  - Счетчики циклов, витки которых распределяются между нитями при помощи конструкций FOR и PARALLEL FOR.

Классы переменных

```
double Array1[100];
int main() {
    int Array2[100];
#pragma omp parallel
   {  int iam = omp_get_thread_num();
      work(Array2, iam);
      printf("%d\n", Array2[0]);
   }
}
```

```
extern double Array1[100];
void work(int *Array, int iam)
{
    double TempArray[100];
    static int count;
...
}
```



Классы переменных

Можно изменить класс переменной при помощи конструкций:

- SHARED (список переменных)
- PRIVATE (список переменных)
- FIRSTPRIVATE (список переменных)
- LASTPRIVATE (список переменных)
- THREADPRIVATE (список переменных)
- DEFAULT (PRIVATE | SHARED | NONE)

Конструкция PRIVATE

- Конструкция «private(var)» создает локальную копию переменной «var» в каждой из нитей.
  - Значение переменной не инициализировано
  - Приватная копия не связана с оригинальной переменной
  - В OpenMP 2.5 значение переменной «var» не определено после завершения параллельной конструкции

```
#pragma omp parallel for private (i,j,sum)
for (i=0; i< m; i++)
{
   sum = 0.0;
   for (j=0; j< n; j++)
     sum +=b[i][j]*c[j];
   a[i] = sum;
}
```

Конструкция FIRSTPRIVATE

- «firstprivate» является специальным случаем «private».
  - Инициализирует каждую приватную копию соответствующим значением из главной (master) нити.

```
BOOL FirstTime=TRUE;
#pragma omp parallel for firstprivate(FirstTime)
for (row=0; row<height; row++)
{
  if (FirstTime == TRUE)  { FirstTime = FALSE;   FirstWork (row); }
 AnotherWork (row);
}
```

Конструкция LASTPRIVATE

- lastprivate передает значение приватной переменной, посчитанной на последней итерации в глобальную переменную.

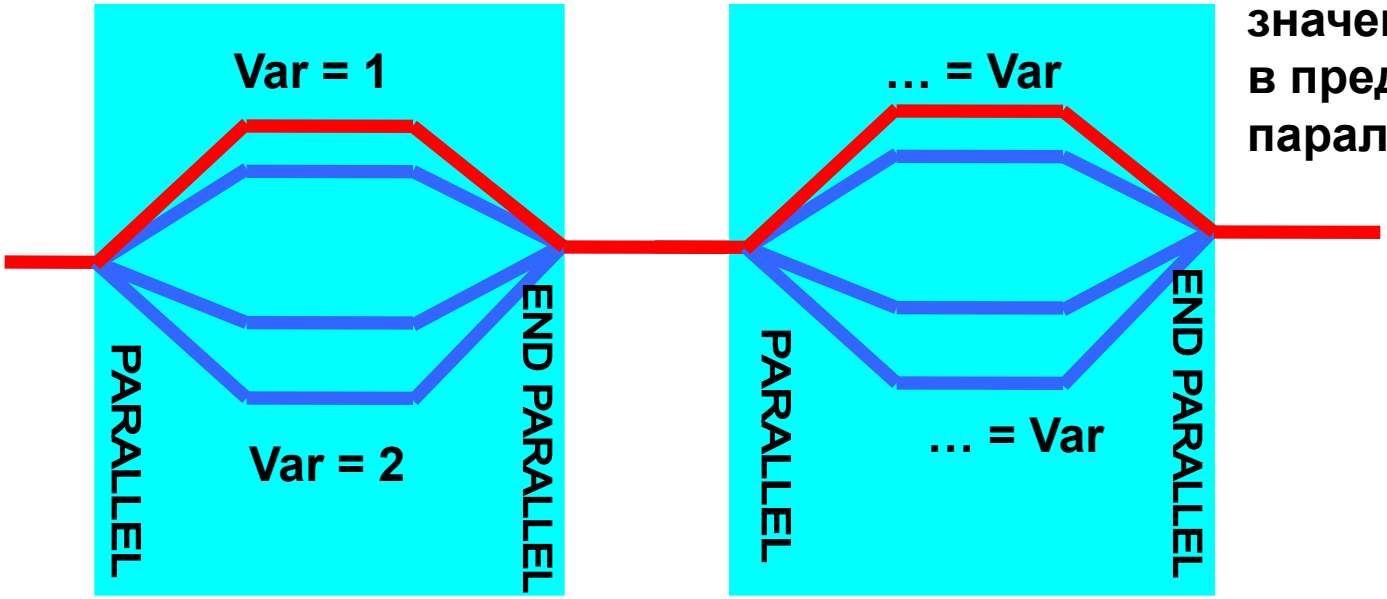
```
int i;
#pragma omp parallel
{
  #pragma omp for lastprivate(i)
    for (i=0; i<n-1; i++)
      a[i] = b[i] + b[i+1];

}
a[i]=b[i];   /*i == n-1*/
```

Конструкция THREADPRIVATE

- Отличается от применения конструкции PRIVATE:
  - с PRIVATE глобальные переменные маскируются
  - THREADPRIVATE сохраняют глобальную область видимости внутри каждой нити

#pragma omp threadprivate (Var)

Если количество нитей не изменилось, то каждая нить получит значение, посчитанное в предыдущей параллельной области.



Конструкция DEFAULT

Меняет класс переменной по умолчанию:

- DEFAULT (SHARED) – действует по умолчанию
- DEFAULT (PRIVATE) – есть только в Fortran
- DEFAULT (NONE) – требует определить класс для каждой переменной

```
itotal = 100
#pragma omp parallel
private(np,each)
{
      np = omp_get_num_threads()
      each = itotal/np
……
}
```

```
itotal = 100
#pragma omp parallel default(none)
private(np,each) shared (itotal)
{
      np = omp_get_num_threads()
      each = itotal/np
……
}
```

Литература…

- http://www.openmp.org
- Распределенные системы. Принципы и парадигмы. / Э. Таненбаум, М. ван Стеен. – СПб. Питер, 2003
- Операционные системы распределенных вычислительных систем (распределенные ОС). Крюков Виктор Алексеевич. http://parallel.ru/krukov/index.html

Вопросы?

Вопросы?

Контакты

- **Бахтин В.А.**, кандидат физ.-мат. наук, заведующий сектором, Институт прикладной математики им. М.В. Келдыша РАН

  bakhtin@keldysh.ru