Нижегородский государственный университет им. Н.И.Лобачевского

Факультет Вычислительной математики и кибернетики

Программирование для Intel Xeon Phi

Выполнение программ на Intel Xeon Phi. Модели организации вычислений с использованием Intel Xeon Phi

Линёв А.В.
2014
Архангельск

Содержание

- Программное обеспечение сопроцессора Intel Xeon Phi
- Модели использования сопроцессора Intel Xeon Phi
- Режим выполнения Offload

Программное обеспечение сопроцессора Intel Xeon Phi

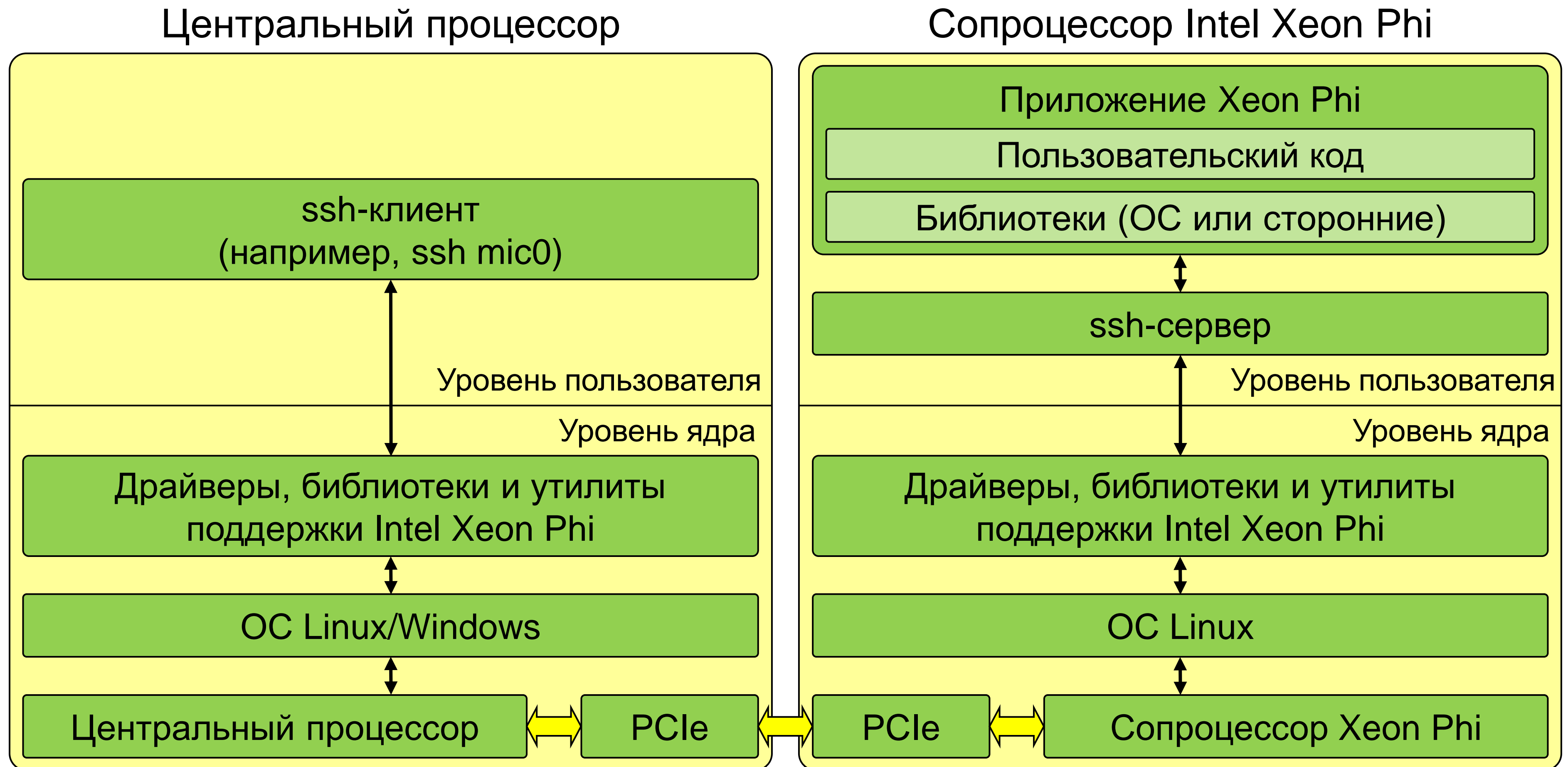
Программное обеспечение сопроцессора Intel Xeon Phi

- Архитектура и состав программного обеспечения для сопроцессора Intel Xeon Phi ориентированы на выполнение высокопроизводительных приложений, способных максимально использовать возможность одновременного выполнения сотен потоков
- Встроенное ПО позволяет использовать его в системах с шиной PCI Express, работающих под управлением операционных систем Linux или Windows

Операционная система сопроцессора Intel Xeon Phi…

- С точки зрения базовой ОС сопроцессор представляет собой отдельный вычислительный SMP-домен, работающий под управлением собственной операционной системы и слабо связанный с основными процессорами системы
- ОС сопроцессора базируется на стандартном ядре Linux, в которое были внесены минимально возможные изменения, требуемые для поддержки новой архитектуры

Операционная система сопроцессора Intel Xeon Phi…

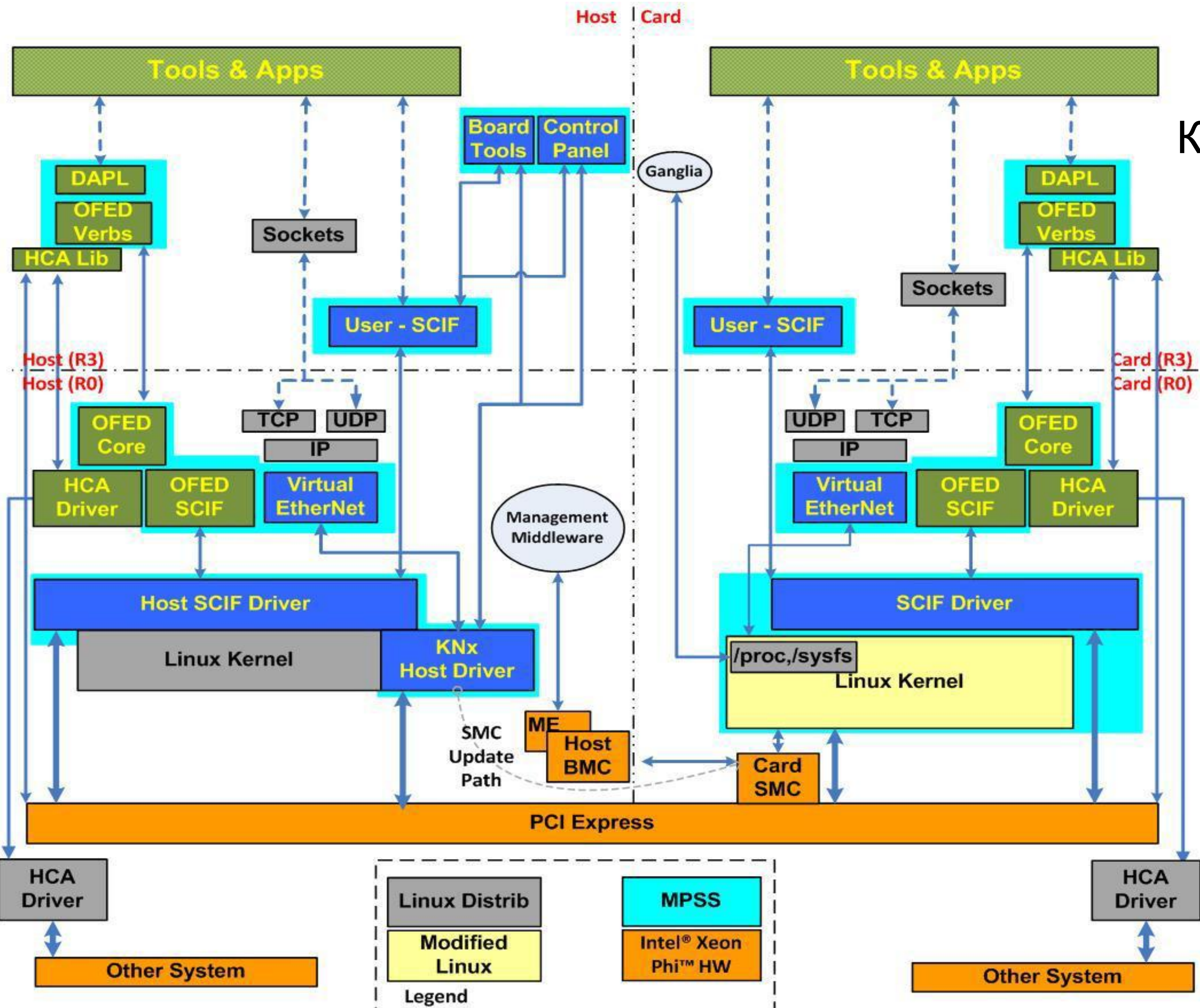


Операционная система сопроцессора Intel Xeon Phi

- Процесс старта и загрузки ОС сопроцессора похож на загрузку ОС базовой системы
  - Bootstrap начинает выполняться при подаче электрического питания на сопроцессор или его перезагрузке
    - fboot0 расположен в ROM сопроцессора и не может быть изменена
    - fboot1 располагается во флеш-памяти и допускает обновление; выполняет загрузку операционной системы сопроцессора и передачу управления ее загрузчику (Linux Loader)

Intel Manycore Platform Software Stack (MPSS)…

- Intel MIC Architecture Manycore Platform Software Stack (MPSS) включает программные компоненты, необходимые для управления сопроцессором и обеспечения взаимодействия с ним
- Взаимодействие между базовой системой и сопроцессорами можно осуществлять несколькими различными способами, доступны реализации ряда стандартных и специализированных API
  - сокеты TCP/IP
  - MPI
  - OpenCL
  - SCIF API (Symmetric Communication Interface API)



Компоненты Intel Manycore Platform Software Stack (MPSS)

Intel Xeon Phi Coprocessor System Software Developers Guide

Intel Manycore Platform Software Stack (MPSS)…

- Симметричный коммуникационный интерфейс (Symmetric Communication Interface, SCIF) – базовый механизм взаимодействия между процессорами основной системы и сопроцессором
  - Взаимодействие между парой клиентов SCIF основано на прямом доступе к памяти друг друга, в частности, взаимодействие между двумя сопроцессорами производится без использования оперативной памяти основной системы

Intel Manycore Platform Software Stack (MPSS)…

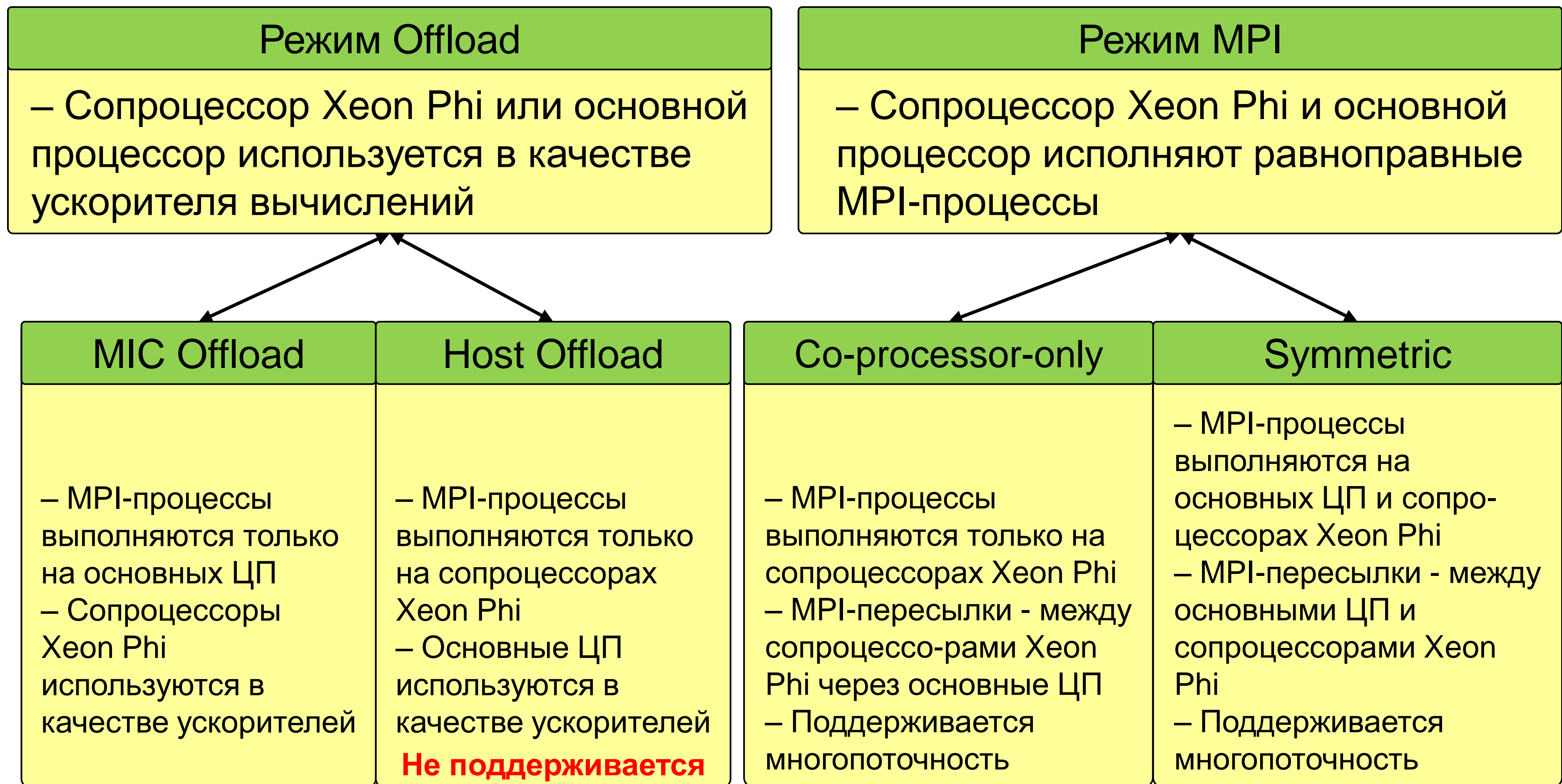
- OFED (OpenFabrics Enterprise Distribution) – комплекс программных средств для использования RDMA (Remote Direct Memory Access, удаленный прямой доступ к памяти) и выполнения приема/передачи данных без вызовов ядра ОС
  - широко используется в приложениях, требующих эффективной работы с сетью и хранилищами данных, и при организации параллельных вычислений
  - является предпочтительным механизмом передачи для библиотеки Intel MPI
- Поддержка Ganglia
  - предоставляется стандартный интерфейс доступа к данным мониторинга на всех сопроцессорах Intel Xeon Phi (виртуальные файловые системы proc или sysfs)
  - образцы плагинов для сбора дополнительных метрик и настроек системы мониторинга
  - каждый сопроцессор представляется в системе мониторинга как отдельный узел

Intel Manycore Platform Software Stack (MPSS)

- Intel MPI для архитектуры MIC поддерживает работу только с системой управления Hydra
  – Каждый узел и каждый сопроцессор идентифицируются уникальным символьным именем или IP-адресом, что позволяет выбрать для каждого процесса исполняемый файл, соответствующий архитектуре, на которой он будет выполняться
- Intel Debugger (idb) позволяет отлаживать как приложения, работающие только на сопроцессоре, так и приложения, использующие и центральный процессор, и сопроцессор
  – можно использовать GNU Project Debugger (gdb)

Модели использования сопроцессора Intel Xeon Phi

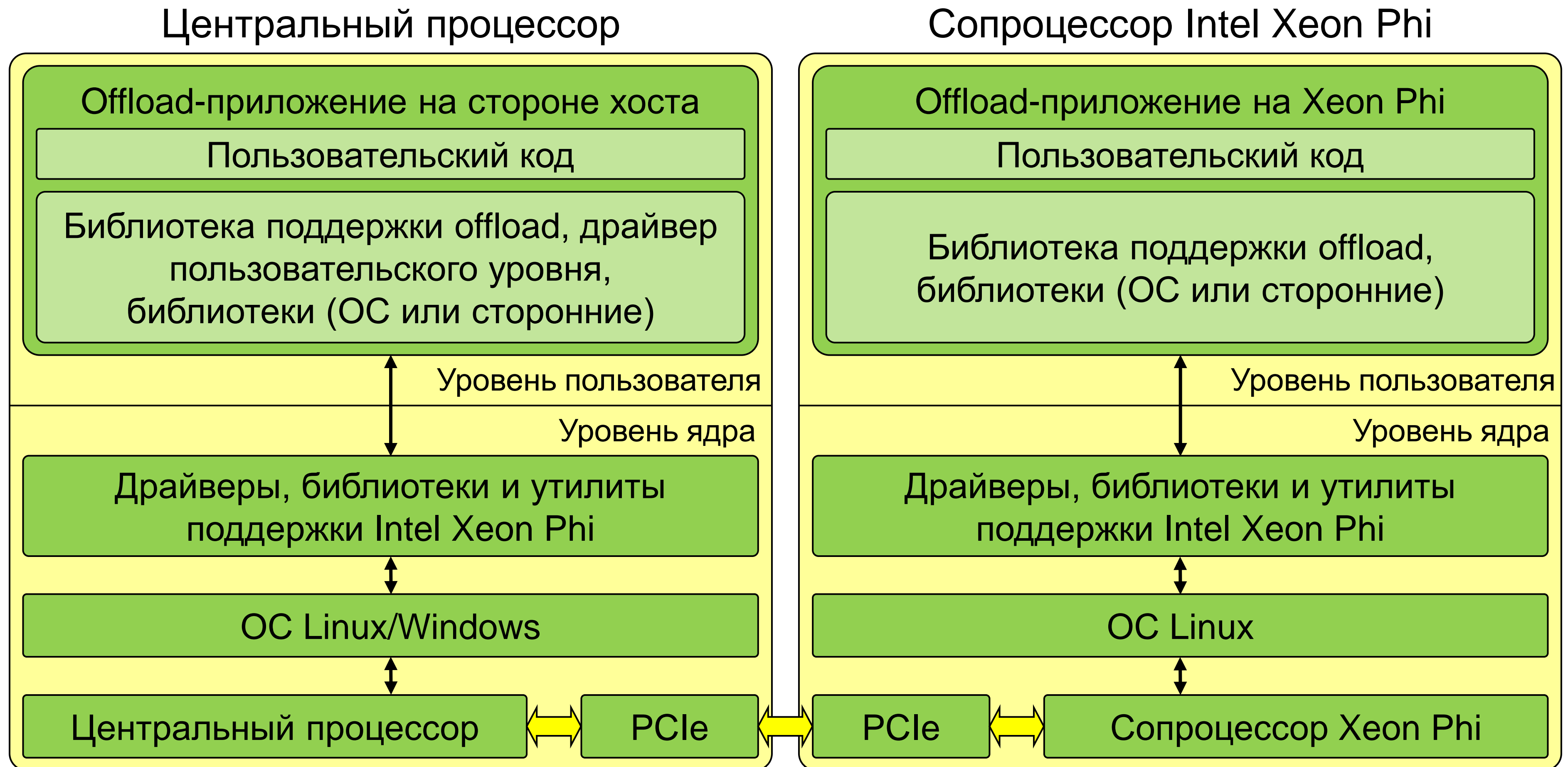
Режимы и модели использования сопроцессора Intel Xeon Phi…



Режимы и модели использования сопроцессора Intel Xeon Phi…

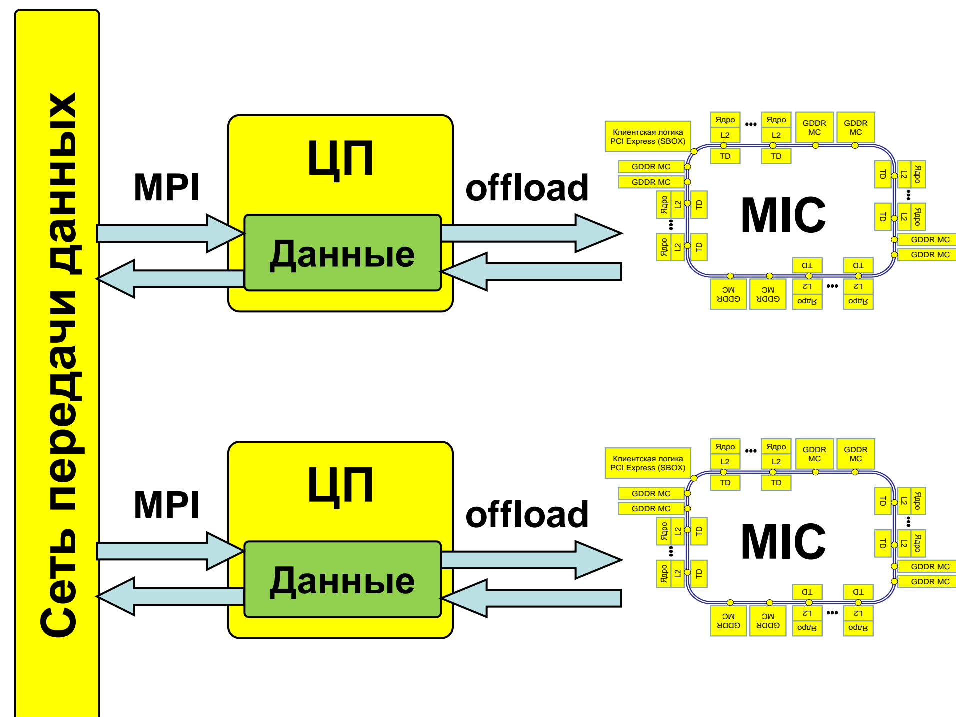
- При работе в режиме Offload используется один из следующих подходов
  - Процессы MPI выполняются на процессорах Xeon базовой системы, для вычислений может использоваться запуск функций на сопроцессоре Xeon Phi (MIC Offload или просто Offload)
    - Поддерживается компиляторами C, C++, Fortran для архитектуры MIC, библиотекой Intel MKL, а также Intel MPI for Linux начиная с версии 4.0. Update 3
  - Процессы MPI выполняются на процессорах Xeon Phi с возможным запуском выполнения функций на процессорах Xeon базовой системы (Host Offload)
    - Не поддерживается текущими версиями компиляторов и библиотек

Исполнение в режиме Offload на одном узле



Режим выполнения Offload

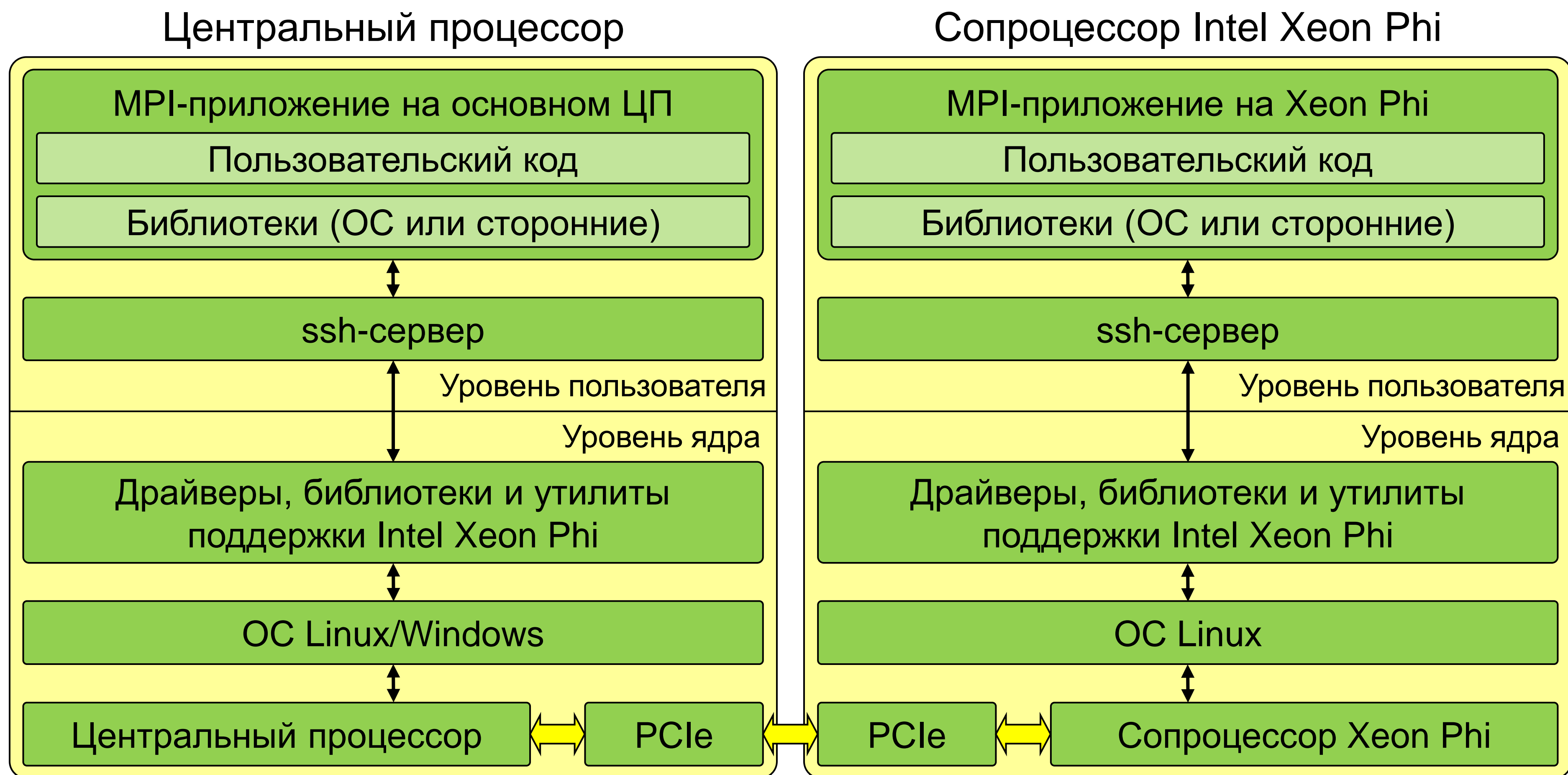
- MPI-процессы выполняются на процессорах базовой системы
- Для использования вычислительных возможностей Xeon Phi используется выгрузка и выполнение функций на сопроцессоре
  - вызов функций MPI в выгруженном коде не поддерживаться



Режимы и модели использования сопроцессора Intel Xeon Phi…

- В режиме MPI базовая система и каждый сопроцессор Intel Xeon Phi рассматриваются как отдельные равноправные узлы, и процессы MPI могут выполняться на процессорах Xeon базовых систем и сопроцессорах Xeon Phi в произвольных сочетаниях

Исполнение в режиме MPI на одном узле

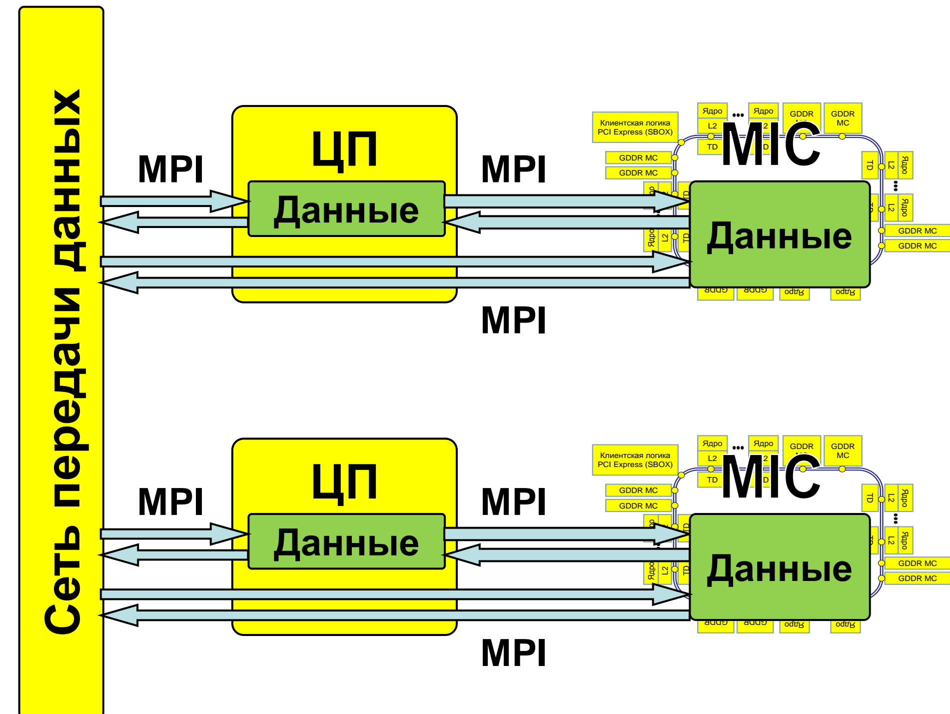


Режимы и модели использования сопроцессора Intel Xeon Phi…

- Три основные модели выполнения в режиме MPI
  - Модель симметричного выполнения (Symmetric model)
    - MPI-процессы выполняются как на процессорах базовой системы, так и на сопроцессорах
  - Модель использования только сопроцессоров (Coprocessor-only model)
    - Все MPI-процессы выполняются на сопроцессорах
  - Модель использования только процессоров базовой системы (Host-only model)
    - MPI-процессы выполняются на процессорах базовой системы, сопроцессоры не используются

Модель симметричного выполнения

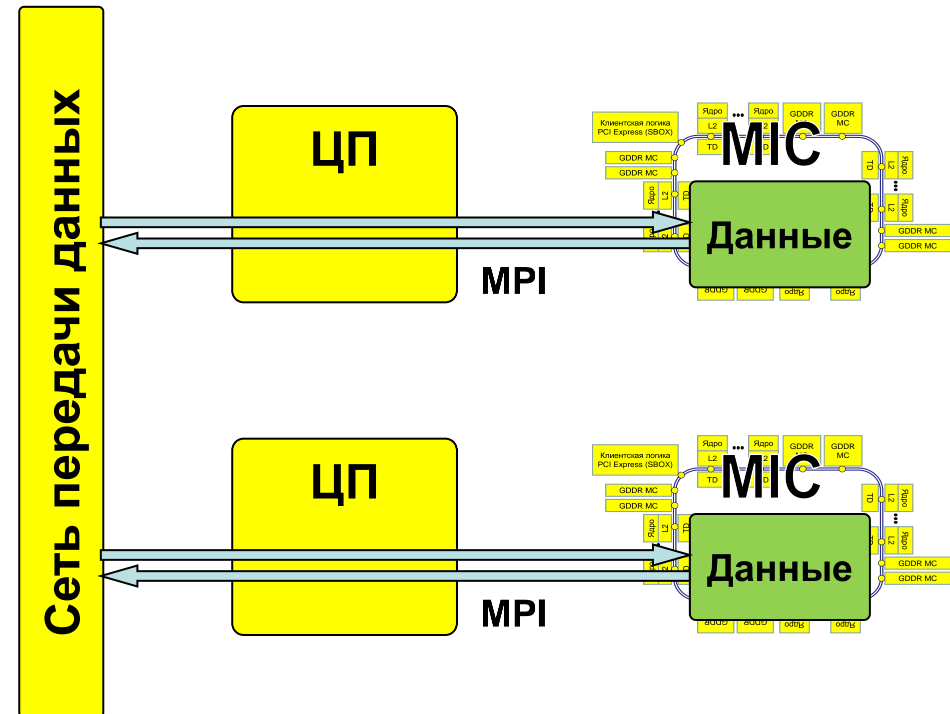
- MPI-процессы выполняются как на процессорах базовой системы, так и на сопроцессорах
- Передача сообщений между процессорами базовой системы, в пределах сопроцессора и между сопроцессором и процессорами базовой системы может выполняться через механизмы разделяемой памяти или протокола tcp
  - по умолчанию tcp
  - I_MPI_SSHM_SCIF={enable|yes|on|1}




```
mpiexec.hydra -host $(hostname) -n 4 -env OMP_NUM_THREADS 4 ./test.exe.host
-host mic0 -n 2 -env OMP_NUM_THREADS 16 -wdir /tmp /tmp/test.exe.mic
```

Модель использования только сопроцессоров

- Native model для Xeon Phi
- Приложение, библиотека MPI и другие необходимые библиотеки загружаются на сопроцессор для выполнения
- Запуск приложения может быть произведен как из базовой ОС, так и из ОС сопроцессора



Инструменты разработчика

- Средства разработки «Intel Parallel Studio XE 2013», «Intel Cluster Studio XE 2013», «Intel SDK for OpenCL Applications XE 2013 Beta», gcc (в настоящий момент gcc не поддерживает векторные инструкции Xeon Phi),…
- Библиотеки Intel Math Kernel Library (Intel MKL), Intel Threading Building Blocks (Intel TBB), Intel Integrated Performance Primitive (Intel IPP), Intel MPI for Linux, MPICH2, Boost,…
- Отладчики (Intel Debugger, gdb, totalview), профилировщики, средства виртуализации (xen),…

Режим выполнения Offload

Последовательное синхронное выполнение…

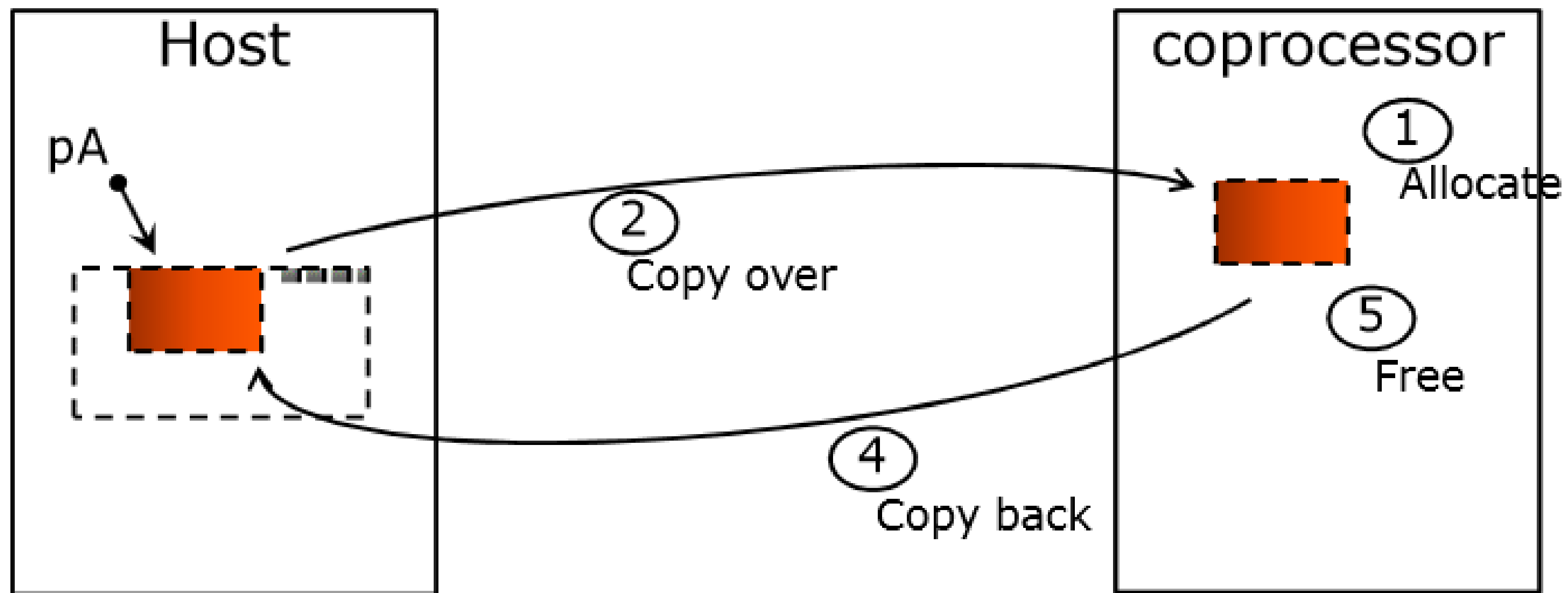
- Блок программы, который следует выполнить на сопроцессоре, указывается по-средством директивы #pragma offload target(mic)
- Операторы in, out, inout определяют необходимость и направление передачи данных между памятью хоста и сопроцессора
- По умолчанию все переменные, объявленные вне offload-блока, перед началом его выполнения копируются на сопроцессор, а по окончании выполнения копируются назад в память базовой системы

Последовательное синхронное выполнение...

```
float Sum(float *Data, int Size){
   float Ret = 0.0f;
#pragma offload target(mic)
  in(Data:length(Size))
   for (int i = 0; i < Size; i ++){
      Ret += Data[i];
   }
   return Ret;
}
```

// Offload-код будет выполняться последовательно на
// одном ядре сопроцессора.
// Переменная Ret перед началом выполнения offload-кода
// будет скопирована из памяти хост-системы в память
// сопроцессора, а по окончании – обратно.

Последовательное синхронное выполнение…



Последовательное синхронное выполнение

- #pragma offload target(mic)
  - При выполнении первой директивы Offload – если MIC доступен, на него загружается MIC-версия программы
  - При выполнении директив Offload – если MIC доступен, оператор/блок выполняется на MIC, если недоступен – оператор/блок выполняется на основном процессоре хоста
  - При завершении программы на хосте происходит выгрузка программы с MIC

Последовательное синхронное выполнение

- #pragma offload target(mic:N)
  - Код будет выполнен на сопроцессоре с номером (n % <ОбщееЧислоXeonPhi>)
  - Если сопроцессор недоступен – возникнет ошибка времени исполнения

Обзор возможностей Pragmas and directives

- Offload pragma
  - #pragma offload <clauses> <statement>
  - Следующий оператор сможет выполняться на CPU или на MIC
- Квалификатор функций/переменых
  - __attribute__((target(mic)))
  - Скомпилировать функцию для CPU и для MIC, разместить переменную как на CPU, так и на MIC
- Квалификатор для блока
  - #pragma offload_attribute(push, target(mic))
  - …
  - #pragma offload_attribute(pop)
  - Скомпилировать весь блок для CPU и MIC

Обзор возможностей Options

- Выбор сопроцессора при наличии нескольких
  - target(mic[:unit])
- Условный offload
  - if (condition) / manadatory
- Входные параметры
  - in(var-list modifiersopt)
- Выходные параметры
  - out(var-list modifiersopt)
- Входные/выходные параметры
  - inout(var-list modifiersopt)
- Не-копируемые данные
  - nocopy(var-list modifiersopt)
  - Локальные данные MIC

Обзор возможностей Modifiers

- Явное указание числа копируемых данных
  - length(N)
- Выделение памяти на сопроцессоре
  - alloc_if(bool)
  - Выделить память при данном offload(default: TRUE)
- Освобождение памяти на сопроцессоре
  - free_if(bool)
  - Освободить память при данном offload(default: TRUE)
- Управление выравниванием на сопроцессоре
  - align(N bytes)
- Частичное выделение памяти для массива, перемещение переменных
  - alloc(array-slice)
  - into(var-expr)
  - Копирование данных в другие переменные/индексы массива

Обзор возможностей
Ограничения на копируемые данные

- В offload-секцию кода могут передаваться скалярные переменные, массивы и структуры, допускающие побитовое копирование
  - Нельзя передавать указатели или структуры/массивы, содержащие указатели
  - Нельзя передавать классы C++ (за исключением простейших)

Явная схема работы с памятью: статическая память

```
__attribute__((target(mic))) void func(float* a,
    float* b, int count, float c, float d)
{
    #pragma omp parallel for
    for (int i = 0; i < count; ++i)
    {
        a[i] = b[i]*c + d;
    }
}

int main()
{
    const int count = 100;
    float a[count], b[count], c, d;
    …
    #pragma offload target(mic) in(b) out(a)
        func(a, b, count, c, d);
    …
}
```

Обзор возможностей alloc_if(), free_if()

- По умолчанию переменные на сопроцессоре создаются и уничтожаются при каждом offload-вызове
- Дополнительные управляющие макросы
  - #define ALLOC alloc_if(1)
  - #define FREE free_if(1)
  - #define RETAIN free_if(0)
  - #define REUSE alloc_if(0)
- Создать переменную и сохранить для следующих вызовов
  - #pragma offload target(mic) in(p:length(n) ALLOC RETAIN)
- Повторно использовать переменную и сохранить для следующих вызовов
  - #pragma offload target(mic) in(p:length(n) REUSE RETAIN)
- Повторно использовать переменную и уничтожить
  - #pragma offload target(mic) in(p:length(n) REUSE FREE)

Явная схема работы с памятью: динамическая память…

```
#define ALLOC alloc_if(1) free_if(0)
#define FREE alloc_if(0) free_if(1)
#define REUSE alloc_if(0) free_if(0)

void f()
{
    int *p = (int *)malloc(100*sizeof(int));
    // Memory is allocated for p,
    // data is sent from CPU and retained
    #pragma offload target(mic:0) in(p[0:100] : ALLOC)
    { p[6] = 66; }
    …
    // Memory for p reused from previous offload
    // and retained once again
    // Fresh data is sent into the memory
    #pragma offload target(mic:0) in(p[0:100] : REUSE)
    { p[6] = 66; }
    …
    // Memory for p reused from previous offload,
    // freed after this offload.
    // Final data is pulled from coprocessor to CPU
    #pragma offload target(mic:0) out(p[0:100] : FREE)
    { p[7] = 77; }
    …
}
```

Явная схема работы с памятью: динамическая память

```
void f()
{
    int *p;
    …
    // The nocopy clause ensures CPU values pointed to by p
    // are not transferred to coprocessor
    #pragma offload target(mic:0) nocopy(p)
    {
        // Allocate dynamic memory for p on coprocessor
        p = (int *)malloc(100);
        p[0] = 77;
        …
    }
    ..
    // The nocopy clause ensures p is not altered
    // by the offload process
    #pragma offload target(mic:0) nocopy(p)
    {
        // Reuse dynamic memory pointed to by p
        … = p[0]; // Will be 77
    }

}
```

Явная работа с памятью: одновременное выполнение на процессоре и сопроцессоре

```
double __attribute__((target(mic)))
myworkload(double input)
{
    // do something useful here
    return result;
}

int main(void)
{
    //…. Initialize variables
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            #pragma offload target(mic)
                result1= myworkload(input1);
        }
        #pragma omp section
        {
            result2= myworkload(input2);
        }
    }
}
```

Обзор возможностей
Явное копирование памяти

- Для передачи между хостом и сопроцессором сложных структур данных, например, использующих указатели, в языках C/C++ реализована модель «разделяемой памяти»
  - обеспечивает размещение специальным образом маркированных переменных (квалификатор типа _Cilk_shared) по одним и тем же виртуальным адресам на хост-системе и сопроцессоре, а также включает специальные функции для динамического выделения памяти по одним и тем же адресам на хост-системе и сопроцессоре
  - «Разделяемая память» не может быть реализована непосредственным отображением адресов памяти сопроцессора на адреса памяти хост-системы
  - Данный механизм является вариацией обычного вызова offload-кода – при выполнении вызова функции с использованием квалификатора _Cilk_offload определяется, какие изменения произошли в копии, хранящейся в памяти хост-системы, и изменения передаются в память сопроцессора (аналогично при возврате из функции)

```
// Явное копирование памяти
float * _Cilk_shared Data;

_Cilk_shared float MIC_Sum(int Size) {
   float Result;
   for (int i = 0; i < Size; i++){
      Result += Data[i];
   }
   return Result;
}

int main(){
   size_t Size = 1000000;
   int MemSize;
   MemSize = Size * sizeof(float);
   Data = (_Cilk_shared float *) _Offload_shared_malloc (MemSize);
   for (int i = 0; i < Size; i++){
      Data[i] = i;
   }
   _Cilk_offload MIC_Sum(Size);
   _Offload_shared_free(Data);
   return 0;
}
```

Последовательное синхронное выполнение с векторизацией

```
float Sum(float *Data, int Size){
   float Ret = 0.0f;
#pragma offload target(mic)
  in(Data:length(Size))
   //Intel Cilk Plus Extended Array Notation
   Ret = __sec_reduce_add(Data[0:Size]);
   return Ret;
}
```

// Компилятор Intel по умолчанию выполняет
// векторизацию кода.
// Offload-код будет выполняться последовательно на
// одном ядре сопроцессора с использованием векторных
// вычислений, выполняя по 16 операций сложения за одну
// инструкцию.

Последовательное асинхронное выполнение…

- При использовании режима Offload можно использовать технику двойной буферизации, обеспечивающую одновременное выполнение offload-функции и передачу на сопроцессор входных данных для следующего вызова и/или передачу выходных данных в память основной системы для предыдущего вызова

```
LoadDataBlock(0);
for( i = 0; i < N-1; i ++ ){
  LoadDataBlock(i+1);
  ProcessBlock(i);
}
ProcessBlock(N-1);
```

- (см. …/C++/mic_samples/intro_sampleC/sampleC13.c)
  - #pragma offload target(mic) inout(A:length(2000))
  - #pragma offload_transfer target(mic) in(a: length(2000)) signal(a)

Последовательное асинхронное выполнение…

```
int main(int argc, char* argv[])
{
    // Allocate & initialize in1, res1,
    // in2, res2 on host
    #pragma offload_transfer target(mic:0) in(cnt)\
        nocopy(in1, res1, in2, res2 : length(cnt) \
        alloc_if(1) free_if(0))

    do_async_in();

    // Free MIC memory
    #pragma offload_transfer target(mic:0) \
        nocopy(in1, res1, in2, res2 : length(cnt) \
        alloc_if(0) free_if(1))

    return 0;
}
```

Последовательное асинхронное выполнение…

```
void do_async_in()
{
    float lsum;
    int i;
    lsum = 0.0f;

    #pragma offload_transfer target(mic:0) \
        in(in1 : length(cnt) \
        alloc_if(0) free_if(0)) signal(in1)
```

Последовательное асинхронное выполнение…

```
for (i = 0; i < iter; i++)
    {
        if (i % 2 == 0)
        {
            #pragma offload_transfer target(mic:0) \
                if(i !=iter - 1) in(in2 : length(cnt) \
                alloc_if(0) free_if(0)) signal(in2)

            #pragma offload target(mic:0) nocopy(in1) \
                wait(in1) out(res1 : length(cnt) \
                alloc_if(0) free_if(0))
            {
                compute(in1, res1);
            }

            lsum = lsum + sum_array(res1);
        }
```

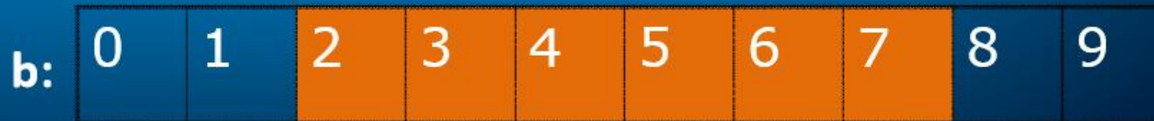
Последовательное асинхронное выполнение…

```
            else
            {
                  #pragma offload_transfer target(mic:0) \
                        if(i != iter - 1) in(in1 : length(cnt) \
                        alloc_if(0) free_if(0))  signal(in1)

                  #pragma offload target(mic:0) nocopy(in2) \
                        wait(in2) out(res2 : length(cnt) \
                        alloc_if(0) free_if(0))
                  {
                        compute(in2, res2);
                  }

                  lsum = lsum + sum_array(res2);
            }
      } // for
} // do_async_in()
```

Технология Array Notation…

- С помощью выражения ***A[:]*** задается весь массив A (размер массива определяется на этапе компиляции, а значит должен быть константным)
- Выражение ***A[start_index : length]*** задает отрезок массива, начиная со *start_index* длиной *length*

```
float b[10];
..
  = b[2:6];
// section operands can be variables also
..
```



Технология Array Notation…

- Выражение ***A[start_index : length : stride]*** говорит о том, что мы хотим использовать каждый *stride* элемент массива, начиная со *start_index*. Количество таких элементов должно быть равно *length*

```
float d[10];
..
  = d[0:3:2];
..
```

d: 0 1 2 3 4 5 6 7 8 9

- Поддерживаются многомерные массивы

Технология Array Notation…

- Возможно использование операторов языков C/C++:

```
d[:] = a[:] + (b[:]*c[:]);
```

- Возможна передача массивов в качестве аргументов функции. При этом вызов функции осуществляется для каждого заданного элемента массива:

```
b[:] = func(a[:]);
```

- Поддерживается операция редукции для сложения, минимума, максимума и т.п.:

```
sum = __sec_reduce_add(a[:]);
```

Технология Array Notation…

- Поддерживаются условные операторы if-then-else:

```
if (mask[:])
{
    a[:] = b[:];
}
```

- Поддерживаются операции типа scatter/gather, с помощью которых можно собрать определенные элементы одного массива в другой (собрать разрозненные элементы в один непрерывный массив), и наоборот:

```
c[:] = a[b[:]]; //gather
a[b[:]] = c[:]; //scatter
```

Технология Array Notation…

- Поддерживаются операции сдвига. Операция ***shift*** сдвигает элементы массива на *shift_val* позиций влево/вправо, освободившиеся элементы заполняются значением *fill_val*. Операция ***rotate*** обеспечивает циклический сдвиг элементов влево/вправо на ***rotate_val*** позиций. Результат работы этих функций записывается в новый массив:

```
b[:] = __sec_shift_right(a[:], shift_val, fill_val);
b[:] = __sec_shift_left(a[:], shift_val, fill_val);
b[:] = __sec_rotate_right(a[:], rotate_val);
b[:] = __sec_rotate_left(a[:], rotate_val);
```

Технология Array Notation…

- Размер массива должен быть известен на стадии компиляции. Если используется динамический массив, то при использовании данной нотации необходимо явно указывать начальную позицию (*start_index*) и длину (*length*) отрезка массива.
- В случае если векторизация кода невозможна, будет сгенерирован обычный цикл.
- Оптимальный векторный код будет получен только в случае работы с выровненными данными.
- Требуется соответствие рангов и длин массивов в рамках одной операции.

Технология Array Notation

- Скалярный код:

```
float dot_product(unsigned int size, float *A, float *B)
{
    int i;
    float dp=0.0f;
    for (i=0; i<size; i++)
    {
        dp += A[i] * B[i];
    }

    return dp;
}
```

- Векторный код:

```
float dot_product(unsigned int size, float A[size], float
B[size])
{
    return __sec_reduce_add(A[:] * B[:]);
}
```

Заключение

- Выполнения программ на Intel Xeon Phi
  - Режим MIC Offload
  - Модель использования только сопроцессоров в режиме MPI
  - Модель симметричного выполнения в режиме MPI
- Несмотря на то, что большая часть лекции посвящена режиму Offload (как обладающему существенной спецификой), рекомендуется при разработке программ для Intel Xeon Phi использовать режим MPI

Литература

- Intel and Third Party Tools and Libraries available with support for Intel® Xeon Phi™ Coprocessor
  - [http://software.intel.com/en-us/articles/intel-and-third-party-tools-and-libraries-available-with-support-for-intelr-xeon-phitm]
- User and Reference Guide for the Intel® C++ Compiler
  - [http://software.intel.com/en-us/compiler_14.0_ug_c]
- Reinders J. An Overview of Programming for Intel Xeon processors and Intel Xeon Phi coprocessors.
  - [http://software.intel.com/en-us/blogs/2012/11/14/an-overview-of-programming-for-intel-xeon-processors-and-intel-xeon-phi]
- Loc Q Nguyen et al. Intel Xeon Phi Coprocessor Developer's Quick Start Guide.
  - [http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-developers-quick-start-guide]
- Intel Xeon Phi Coprocessor System Software Developers Guide.
  - [http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-system-software-developers-guide]
- Rahman R. Intel Xeon Phi Core Micro-architecture
  - [http://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture]
- Robert Reed. An Introduction to the Intel® Xeon Phi™ Coprocessor.