Нижегородский государственный университет им. Н.И. Лобачевского
Факультет вычислительной математики и кибернетики

Образовательный комплекс «Введение в принципы функционирования и применения современных мультиядерных архитектур (на примере Intel Xeon Phi)»

Лабораторная работа №1 Компиляция и запуск приложений на Intel Xeon Phi

Горшков А.В.

При поддержке компании Intel

Нижний Новгород
2013

Содержание

ВВЕДЕНИЕ **3**

1. МЕТОДИЧЕСКИЕ УКАЗАНИЯ **3**

1.1. Цели и задачи работы 3
1.2. Структура работы 4
1.3. Тестовая инфраструктура 4
1.4. Рекомендации по проведению занятий 4

2. МОДЕЛИ ПРОГРАММИРОВАНИЯ ПРИЛОЖЕНИЙ НА INTEL XEON PHI **5**

2.1. Режим offload 6
2.2. Режим исполнения на сопроцессоре 12
2.3. Симметричный режим 19

3. ДОПОЛНИТЕЛЬНЫЕ ЗАДАНИЯ **21**

4. ЛИТЕРАТУРА **22**

4.1. Использованные источники информации 22

Введение

Использование ускорителей для решения сложных вычислительных задач сегодня норма. В большинстве случаев для этой цели применяют графические процессоры, которые обладают значительно большей, чем CPU, производительностью. Основная сложность процесса переноса большого программного пакета на GPU заключается в необходимости значительной модификации существующего кода для возможности его последующего выполнения на графическом процессоре. Более того, GPU имеет принципиально иную архитектуру, а значит, для получения хорошей производительности, нужно не просто обеспечить выполнение программы на графическом процессоре, но и выполнять ряд действий по ее оптимизации. А это в свою очередь может привести к необходимости переработки использующихся в программе алгоритмов.

Недавно представленный компанией Intel сопроцессор с архитектурой Intel Many Integrated Core (MIC) на базе x86 ядер CPU позволяет существенно сократить время переноса. Это достигается за счет использования известных инструментов, таких как Intel C/C++ Compiler, и технологий параллельного программирования (OpenMP, TBB, MPI). Более того, для обеспечения работоспособности кода на Intel MIC, часто модификации существующего кода не требуется совсем. Следует, однако, заметить, что для получения хорошей производительности оптимизация программы является необходимым этапом переноса, несмотря на схожесть архитектуры центрального процессора и сопроцессора.

В данной лабораторной работе рассматривается ряд примеров, демонстрирующих процесс переноса кода на сопроцессор Intel Xeon Phi. Рассматриваются основные модели программирования и режимы компиляции и запуска программ.

1. Методические указания

1.1. Цели и задачи работы

Цель данной работы – изучение режимов и способов компиляции и запуска программ на Intel Xeon Phi.

Данная цель предполагает решение следующих основных задач:

1. Изучение моделей программирования на Intel Xeon Phi.
2. Освоение способов компиляции и запуска программ на одном или нескольких сопроцессорах применительно к каждой из рассматриваемых моделей программирования.

1.2. Структура работы

Работа построена следующим образом: дан краткий обзор моделей программирования для сопроцессора Intel Xeon Phi, далее на примерах продемонстрированы подходы к написанию кода в каждом из режимов, а также шаги, которые необходимо выполнить для компиляции и запуска программ. В рамках данной лабораторной работы рассматриваются режим Offload, режим работы только на сопроцессоре и симметричный режим.

1.3. Тестовая инфраструктура

Вычислительные эксперименты проводились с использованием следующей инфраструктуры (табл. 1).

Таблица 1. Тестовая инфраструктура

Процессор	Intel Xeon E5-2690 (2.9 GHz, 8 ядер)
Сопроцессор	Intel Xeon Phi 7110X
Память	64 GB
Операционная система	Linux CentOS 6.2
Компилятор, профилировщик, отладчик	Intel C/C++ Compiler 13

1.4. Рекомендации по проведению занятий

Для выполнения лабораторной работы рекомендуется следующая последовательность действий:

1. Кратко напомнить студентам о моделях программирования на Intel Xeon Phi, дать краткую характеристику каждой из моделей. Обратить внимание на достоинства и недостатки моделей.
2. Рассмотреть пример вывода максимального числа потоков сопроцессора в режиме Offload.
3. Рассмотреть пример простого вычисления скалярного произведения векторов в режиме Offload.
4. Рассмотреть пример простого вычисления скалярного произведения векторов в режиме работы только на сопроцессоре. Рассмотреть MPI версию скалярного произведения векторов.
5. Рассмотреть пример вычисления скалярного произведения векторов в симметричном режиме.

6. Сформулировать выводы, дать задания для самостоятельной работы.

2. Модели программирования приложений на Intel Xeon Phi

Разработка программ для Intel Xeon Phi предполагает выбор одной из моделей программирования, поддерживаемых посредством библиотеки Intel MPI. Существующие модели приведены на Рис. 1.

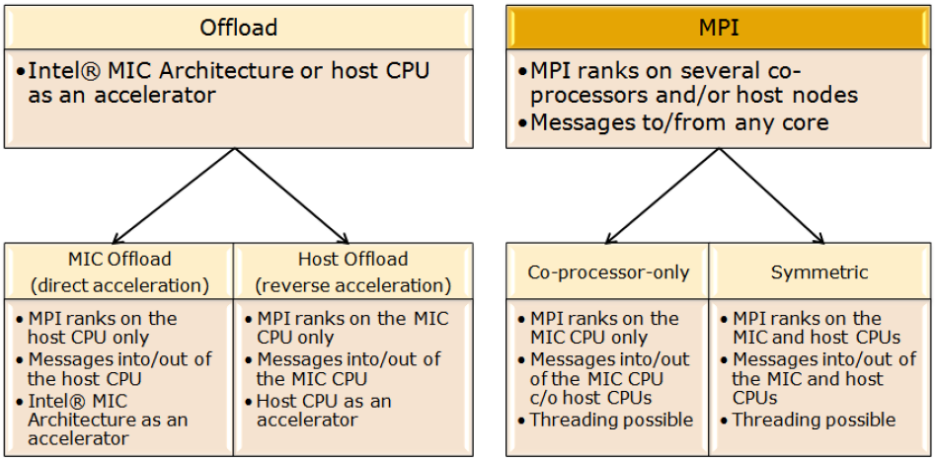

Рис. 1. Модели программирования приложений для Intel Xeon Phi [1]

Offload модель предполагает использование Xeon Phi в режиме сопроцессора, т.е. дополнительного вычислительного устройства, доступ к которому осуществляется с помощью специальных команд (директив) из кода, выполняемого на обычном центральном процессоре. Данный режим поддерживается библиотекой Intel MPI, начиная с версии 4.0 update 3, для операционных систем семейства Linux при условии, что в качестве центрального процессора выступает процессор семейства Intel Xeon (или совместимые процессоры сторонних производителей). При выполнении MPI программы в этом режиме, ранги присваиваются только центральным процессорам.

Теоретически возможен и обратный вариант, когда в роли сопроцессора выступает центральный процессор, а основной код работает на Xeon Phi. Однако этот вариант на текущий момент не поддерживается.

Offload модель поддерживается такими продуктами Intel, как компиляторы C/C++ и Fortran, а также Intel Math Kernel Library (MKL).

В режиме MPI как центральный процессор, так и сопроцессор являются отдельными вычислительными узлами и могут взаимодействовать между собой посредством обмена MPI сообщениями. Написание кода для сопро-

цессора в этом случае ничем не отличается от написания MPI программы для CPU. Внимание нужно уделять разве что балансировке нагрузки и особенностям оптимизации кода под архитектуру Intel MIC.

Обычно рассматриваются три MPI модели:

- **Симметричная модель.** Вычисления производятся как на процессоре, так и на сопроцессоре, причем с точки зрения MPI программирования они являются отдельными узлами и обладают собственными рангами. Это наиболее общая модель.
- **Использование только сопроцессора.** Программа исполняется только на сопроцессоре (одном или многих). Каждый сопроцессор обладает собственным MPI рангом. CPU в вычислениях не задействован.
- **Использование только процессора.** Программа исполняется только на центральных процессорах без использования сопроцессоров. По сути, это обычное исполнение MPI программ. В данной лабораторной работе этот режим рассматриваться не будет.

2.1. Режим offload

В данном разделе рассматривается процесс компиляции и запуска программы для Intel Xeon Phi в режиме offload.

2.1.1. Вывод максимального числа потоков сопроцессора

Рассмотрим простую, но полезную задачу: необходимо вывести максимальное число потоков центрального процессора и сопроцессора Xeon Phi.

Для решения задачи будем использовать возможности технологии OpenMP. Создадим файл **main.cpp**, куда будем писать код. Сначала напишем функцию для вывода на консоль доступного числа потоков, которая работала бы на сопроцессоре. Для этого воспользуемся директивой **offload_attribute** [2]:

```
#pragma offload_attribute(push, target(mic))

#include <stdio.h>
#include <omp.h>

void testThreadCount()
{
    int thread_count;

    #pragma omp parallel
    {
        #pragma omp single
```

```
            thread_count = omp_get_num_threads();
    }
    printf("Number of threads: %d\n", thread_count);
}

#pragma offload_attribute(pop)
```

Первый параметр директивы **offload_attribute** может принимать значения **push** либо **pop**, что соответствует началу и концу блока кода, который предназначен для выполнения как на центральном процессоре, так и на сопроцессоре.

Далее напишем код функции **main**:

```
int main()
{
    printf("Intel CPU:\n");
    testThreadCount();

    int number_of_coprocessors =
        _Offload_number_of_devices();

    printf("Intel Xeon Phi:\n");
    printf("Number of coprocessors: %d\n",
        number_of_coprocessors);

    for (int i = 0; i < number_of_coprocessors; ++i)
    {
        #pragma offload target(mic:i)
        {
            testThreadCount();
        }
    }

    return 0;
}
```

Количество сопроцессоров Intel Xeon Phi на данном узле можно узнать с помощью встроенной функции **_Offload_number_of_devices()**.

Новой здесь является директива **offload**. Блок кода, который расположен за ней, будет выполнен на сопроцессоре. Параметр **target(mic:<device_id>)** указывает, на каком именно сопроцессоре должен быть выполнен код. Следует отметить, что номера доступных сопроцессоров лежат в диапазоне от 0 до общего количества Intel Xeon Phi, установленных в рамках данного узла.

Заметим также, что запуск кода на сопроцессоре выполняется в синхронном режиме, т.е. при вызове функции управление передается сопроцессору, а выполнение основной программы блокируется, пока сопроцессор не завершит свою работу.

Для организации асинхронных вычислений нужно использовать несколько CPU потоков, в каждом из которых может использоваться либо отдельный сопроцессор, либо выполняться некие вычисления на процессоре.

Обратите внимание на разницу в директивах **offload** и **offload_attribute**.

С помощью директивы **offload_attribute** дается указание компилятору скомпилировать блок кода специальным образом для возможности его выполнения на Xeon Phi. При этом данный код не обязательно будет исполняться на сопроцессоре. Эту директиву нельзя использовать внутри функции. Обычно она обрамляет код функций и объявление глобальных переменных.

Директива **offload** говорит компилятору, что блок кода, непосредственно следующий за ней, должен быть выполнен на сопроцессоре. И если это не просто вызов функции, то этот блок кода также будет скомпилирован специальным образом.

Теперь скомпилируем написанную программу, используя Intel C/C++ Compiler. Поддержка offload режима добавлена в компиляторы Intel для Linux, начиная с версии 13. Сам процесс компиляции в данном случае ничем не отличается от компиляции кода для центрального процессора:

```
icc -02 -openmp main.cpp -o lab1_thread_test
```

В результате будет создан исполняемый файл **lab1_thread_test**, содержащий в себе как код центрального процессора, так и код сопроцессора.

Для запуска программы необходимо воспользоваться утилитой Hydra Process Manager, входящей в состав Intel MPI:

```
mpiexec.hydra -perhost 1 ./lab1_thread_test
```

После выполнения данной команды будет произведен запуск программы с использованием всех доступных узлу сопроцессоров. Результат работы зависит от количества имеющихся сопроцессоров. Для случая одного Intel Xeon Phi вывод программы приведен на Рис. 2.

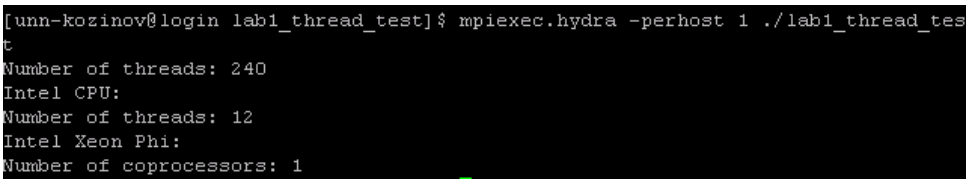

Рис. 2. Результаты запуска программы lab1_thread_test

В случае если работа ведется на кластере с системой управления SLURM, необходимо предварительно получить один из доступных сопроцессоров для монопольного использования:

```
salloc -N 1 --gres=mic:1
```

После этого можно запускать программы на Xeon Phi в течение того промежутка времени, когда вы владеете сопроцессором. Время владения определяется системой управления кластером.

Команда **salloc** позволяет выделить несколько сопроцессоров путем задания соответствующего значения ключа **–N**. Ключ **--gres=mic:1** говорит о необходимости выделить узел кластера с минимум одним сопроцессором Intel Xeon Phi.

В случае успешного завершения появится сообщение о том, что доступ получен. Дополнительно будет указан номер задачи, посредством которой обеспечивается монопольный доступ к сопроцессорам (Рис. 3). Отметим, что в случае отсутствия свободных ресурсов команда **salloc** будет ожидать их появления, блокируя консоль.

```
-sh-4.1$ salloc -N 1 --gres=mic:1
salloc: Pending job allocation 9180
salloc: job 9180 queued and waiting for resources
salloc: job 9180 has been allocated resources
salloc: Granted job allocation 9180
```

Рис. 3. Результат работы команды salloc

Если сопроцессор более не нужен, освободить его можно командой:

```
scancel <номер задачи>
```

Список активных задач выводит команда:

```
squeue
```

2.1.2. Скалярное произведение векторов

Рассмотрим следующую простую задачу: необходимо посчитать скалярное произведение двух векторов.

Создадим файл исходного кода **main.cpp**. Выполним подключение необходимых библиотек:

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

Создадим функцию для подсчета скалярного произведения двух векторов. Вычисление будет производиться параллельно на всех доступных ядрах вычислительного устройства. Обратите внимание, что код функции **dot** не содержит в себе специальных директив для выполнения на Intel Xeon Phi:

```
#pragma offload_attribute(push, target(mic))

float dot(float* a, float* b, int n)
{
    float res = 0;

    #pragma omp parallel for reduction(+: res)
    for (int i = 0; i < n; ++i)
```

```
    {
        res += a[i]*b[i];
    }

    return res;
}

#pragma offload_attribute(pop)
```

Как и прежде, используем директиву **offload_attribute** для того, чтобы сообщить компилятору о необходимости создания копии этого участка кода для исполнения на сопроцессоре.

Напишем функцию **main** для тестирования приведенного выше кода. Прежде всего, объявим используемые переменные и массивы:

```
int main()
{
    int n = 100;
    float* a = new float[n];
    float* b = new float[n];
    float res_cpu, res_mic;
```

Затем инициализируем вектора **a** и **b** случайными числами:

```
    for (int i = 0; i < n; ++i)
    {
        a[i] = (float)rand()/RAND_MAX;
        b[i] = (float)rand()/RAND_MAX;
    }
```

Запустим вычисление скалярного произведения на центральном процессоре и выведем результат на консоль:

```
    res_cpu = dot(a, b, n);

    printf("CPU dot: %f\n", res_cpu);
```

Далее необходимо запустить функцию dot на сопроцессоре. Для этого воспользуемся уже знакомой нам директивой **offload**:

```
    #pragma offload target(mic) in(a[0:n], b[0:n])
    res_mic = dot(a, b, n);

    printf("MIC dot: %f\n", res_mic);
```

Здесь используется дополнительный параметр **in** для указания того, что массивы **a** и **b** размером **n** элементов необходимо скопировать на сопроцессор перед началом вычислений, обратного копирования при этом делать не нужно. После окончания вычислений выведем результат на консоль.

Обратите внимание, что к моменту вызова функции **printf** сопроцессор гарантированно завершит свою работу.

Далее осталось только удалить используемую память центрального процессора. Память на сопроцессоре в данном случае будет освобождена автоматически по окончании работы функции **dot**.

```
    delete[] a;
    delete[] b;

    return 0;
}
```

Скомпилируем и запустим полученный код:

```
icc -02 -openmp main.cpp -o lab1_dot_offload
mpiexec.hydra -perhost 1 ./lab1_dot_offload
```

Результаты работы программы приведены на Рис. 4.

```
[unn-kozinov@login lab1_dot_offload]$ icc -O2 -openmp main.cpp -o lab1_dot_offlo
ad
[unn-kozinov@login lab1_dot_offload]$ mpiexec.hydra -perhost 1 ./lab1_dot_offloa
d
CPU dot: 27.259979
MIC dot: 27.259979
```

Рис. 4. Результаты работы программы lab1_dot_offload

В заключение данного раздела покажем, как осуществить запуск offload программы в пакетном режиме без предварительного выделения ресурсов (сопроцессоров). Этот же способ подходит для запуска MPI программы в режиме offload. Будем использовать возможности системы управления кластером SLURM.

Прежде всего, необходимо написать скрипт запуска. Пусть **X** – число MPI процессов, выполняемых на одном узле кластера, а **Y** – числа узлов. Тогда скрипт будет таким:

```
#!/bin/sh
mpiexec.hydra -perhost X -n X*Y ./offload_program
```

Для постановки задачи в очередь нужно выполнить команды:

```
module load launcher/intel
sbatch -N Y ./run.sh
```

где **run.sh** – имя приведенного выше скрипта. После выполнения этих команд задача будет поставлена в очередь, а после выполнения результаты работы программы будут записаны в файл **slurm-<номер задачи>.out**. Подробнее о работе команды **sbatch** можно узнать из следующих разделов.

2.2. Режим исполнения на сопроцессоре

Модель программирования в «родном» режиме сопроцессора предполагает исполнение кода только на Intel Xeon Phi вообще без использования центрального процессора. С точки зрение написания кода эта модель ничем не отличается от программирования под CPU, различия видны только на этапах компиляции и запуска.

2.2.1. Скалярное произведение векторов на одном сопроцессоре

Запустим рассмотренную выше задачу скалярного умножения векторов на сопроцессоре. Исходный код программы приведен ниже:

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

float dot(float* a, float* b, int n)
{
    float res = 0;

    #pragma omp parallel for reduction(+: res)
    for (int i = 0; i < n; ++i)
    {
        res += a[i]*b[i];
    }

    return res;
}

int main()
{
    int n = 100;
    float* a = new float[n];
    float* b = new float[n];
    float res;

    for (int i = 0; i < n; ++i)
    {
        a[i] = (float)rand()/RAND_MAX;
        b[i] = (float)rand()/RAND_MAX;
    }

    res = dot(a, b, n);
    printf("Result: %f\n", res);

    delete[] a;
    delete[] b;

    return 0;
```

```
}
```

Как видите, код ничем не отличается от «обычного», предназначенного для центрального процессора.

Для компиляции этой программы необходимо выполнить команду:

```
icc -02 –openmp -mmic main.cpp –o lab1_dot_native.mic
```

Обратите внимание на новый ключ компиляции, **-mmic**, который говорит о том, что компиляция будет производиться для сопроцессора. Полученный в результате исполняемый файл может быть запущен только на устройствах с архитектурой MIC, на центральном процессоре он работать не будет.

Для запуска программы можно скопировать исполняемый файл на сопроцессор и запустить его там обычным способом. А можно воспользоваться утилитой MPI Hydra:

```
mpiexec.hydra –host mic0 –n 1 –perhost 1
./lab1_dot_native.mic
```

Здесь параметр **–host** должен указывать на имя сопроцессора (сопроцессоры в данной схеме являются отдельными MPI узлами), **-n** – общее число процессов, **-perhost** – число процессов на узел.

Для запуска программы на кластере с системой управления кластером SLURM необходимо воспользоваться командой **sbatch**:

```
sbatch –N 1 --gres=mic:2 native_run.sh ./lab1_dot_native
```

Параметр **–N** задает число узлов, на которых будет выполнена ваша программа. Ключ **--gres=mic:2** говорит о необходимости выделить для расчетов узел кластера с минимум двумя сопроцессором Intel Xeon Phi. Имя логического раздела с узлами кластера, где установлены сопроцессоры, можно задать ключом **–p**. Обратите внимание, что исполняемый файл задается без расширения **.mic**.

Число MIC процессов на узел по умолчанию зависит от настроек конкретного кластера, обычно оно равно числу доступных сопроцессоров на узле. Т.е. в данном случае программа будет выполнена в два процесса, каждый из которых займет свой сопроцессор.

Приведенная выше команда ставит задачу в очередь. При успешном выполнении будет выведено сообщение о постановке в очередь задачи с определенным номером. Посмотреть состояние задачи можно командой:

```
squeue
```

По завершении задачи в директории с исполняемым файлом будет создан файл **slurm-<номер задачи>.out**, куда запишется консольный вывод нашей программы. Посмотреть содержимое этого файла можно командой:

```
cat slurm-<номер задачи>.out
```

Результат работы нашей программы приведен на Рис. 5. Как видите, результат не отличается от скалярного произведения, вычисленного с помощью предыдущей программы.

```
-sh-4.1$ cat slurm-9197.out
Result: 27.259979
Result: 27.259979
```

Рис. 5. Результаты работы программы lab1_dot_native

Скрипт **native_run.sh** установлен в системе, но для его использования необходимо предварительно выполнить команду:

```
module load launcher/mic
```

Эта команда подгружает модуль **launcher/mic**, делая видимыми скрипты запуска задач на сопроцессоре, такие как **native_run.sh**. Загрузить указанный модуль достаточно один раз в том случае, если он еще не используется. Список используемых в текущий момент модулей можно узнать, воспользовавшись командой:

```
module list
```

Модуль остается загруженным на все время текущей сессии.

Рассмотрим также один из возможных способов задания точного числа MIC процессов на узел. Он состоит в использовании специальных переменных окружения. Конкретные имена переменных зависят от настроек того или иного кластера и обычно указаны в соответствующем руководстве пользователя. Пусть, например, число MIC процессов на узел задается переменной окружения **MICperNODE**. Тогда, чтобы запустить программу в 1 MIC процесс на одном узле, нужно выполнить команды:

```
export MICperNODE=1
sbatch -N 1 --gres=mic:2 native_run.sh ./lab1_dot_native
```

После этого программа будет использовать по одному сопроцессору с каждого из узлов кластера.

2.2.2. Скалярное произведение векторов. MPI версия

Теперь рассмотрим задачу вычисления скалярного произведения массивов векторов. Пусть имеется массив векторов **a** и массив векторов **b**. Количество векторов в массивах одинаково. Необходимо вычислить скалярное произведение всех векторов **a[i]** на **b[i]** для всех **i**. В итоге получим скалярный массив произведений **c**, где **c[i]=dot(a[i], b[i])**.

Очевидно, что данная задача является усложнением предыдущей и может использовать параллельную функцию вычисления скалярного произведения двух векторов, реализованную выше. При этом каждое скалярное произведение может быть вычислено независимо от остальных, а значит, здесь

есть еще один ресурс для распараллеливания. Воспользуемся им для одновременного использования нескольких сопроцессоров в рамках MPI программы.

Писать программу начнем с подключения необходимых библиотек:

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

Добавим функцию вычисления скалярного произведения из предыдущего примера:

```
float dot(float* a, float* b, int n)
{
    float res = 0;

    #pragma omp parallel for reduction(+: res)
    for (int i = 0; i < n; ++i)
    {
        res += a[i]*b[i];
    }

    return res;
}
```

Функция **main** будет содержать весь остальной код. Сначала инициализируем библиотеку MPI, введем необходимые переменные, а так же укажем размерность задачи – размер вектора и количество таких векторов:

```
int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);

    float *a, *b, *c;
    float *send_a = NULL, *send_b = NULL, *recv_c = NULL;
    int vector_size = 500;
    int vector_num = 20;
    int portion_size, remainder;
```

Далее запросим информацию о количестве процессов и ранге текущего процесса:

```
    int mpi_rank, mpi_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
```

Для понимания того, на каких узлах будет запущена наша программа, выведем их список:

```
    if (mpi_rank == 0)
    {
        printf("List of nodes:\n");
```

```
    }

    MPI_Barrier(MPI_COMM_WORLD);

    char node_name[MPI_MAX_PROCESSOR_NAME];
    int node_name_length;
    MPI_Get_processor_name(node_name, &node_name_length);
    printf("%s\n", node_name);
```

Приведенный ниже код описывает алгоритм вычисления размеров порции данных для обсчета каждым из доступных процессов. Вводятся два типа порций – **send** для массивов **a** и **b**, **recv** для массива **c**:

```
    portion_size = vector_num/mpi_size;
    remainder = vector_num - portion_size*mpi_size;

    int* send_portion_size = new int[mpi_size];
    int* send_portion_index = new int[mpi_size];
    int send_index = 0;
    for (int i = 0; i < remainder; ++i)
    {
        send_portion_size[i] = (portion_size + 1)
            *vector_size;
        send_portion_index[i] = send_index;
        send_index += send_portion_size[i];
    }
    for (int i = remainder; i < mpi_size; ++i)
    {
        send_portion_size[i] = portion_size*vector_size;
        send_portion_index[i] = send_index;
        send_index += send_portion_size[i];
    }

    int* recv_portion_size = new int[mpi_size];
    int* recv_portion_index = new int[mpi_size];
    int recv_index = 0;
    for (int i = 0; i < remainder; ++i)
    {
        recv_portion_size[i] = portion_size + 1;
        recv_portion_index[i] = recv_index;
        recv_index += recv_portion_size[i];
    }
    for (int i = remainder; i < mpi_size; ++i)
    {
        recv_portion_size[i] = portion_size;
        recv_portion_index[i] = recv_index;
        recv_index += recv_portion_size[i];
    }
```

Далее на главном (нулевом) процессе выделяем память под вектора и инициализируем их случайными числами:

```
if (mpi_rank == 0)
{
    send_a = new float[vector_num*vector_size];
    send_b = new float[vector_num*vector_size];
    recv_c = new float[vector_num];

    for (int i = 0; i < vector_num*vector_size; ++i)
    {
        send_a[i] = (float)rand()/RAND_MAX;
        send_b[i] = (float)rand()/RAND_MAX;
    }
}
```

Следует заметить, что для удобства передачи данных по сети вектора каждого из массивов лежат в памяти последовательно друг за другом.

Следующий шаг – выделение памяти под порции данных и их отправка остальным процессам:

```
a = new float[send_portion_size[mpi_rank]];
b = new float[send_portion_size[mpi_rank]];
c = new float[recv_portion_size[mpi_rank]];

MPI_Scatterv(send_a, send_portion_size,
    send_portion_index, MPI_FLOAT, a,
    send_portion_size[mpi_rank], MPI_FLOAT,
    0, MPI_COMM_WORLD);
MPI_Scatterv(send_b, send_portion_size,
    send_portion_index, MPI_FLOAT,
    b, send_portion_size[mpi_rank], MPI_FLOAT,
    0, MPI_COMM_WORLD);
```

Далее выполняем необходимые расчеты:

```
for (int i = 0; i < recv_portion_size[mpi_rank]; ++i)
{
    c[i] = dot(a + i*vector_size,
        b + i*vector_size, vector_size);
}
```

Принимаем результаты от других процессов:

```
MPI_Gatherv(c, recv_portion_size[mpi_rank], MPI_FLOAT,
    recv_c, recv_portion_size, recv_portion_index,
    MPI_FLOAT, 0, MPI_COMM_WORLD);
```

Выводим результаты на консоль:

```
if (mpi_rank == 0)
{
    printf("Results:\n");
```

```
        for (int i = 0; i < vector_num; ++i)
        {
            printf("%f\n", recv_c[i]);
        }

        delete[] send_a;
        delete[] send_b;
        delete[] recv_c;
    }
```

И освобождаем оставшуюся память:

```
    delete[] a;
    delete[] b;
    delete[] c;

    delete[] send_portion_size;
    delete[] send_portion_index;
    delete[] recv_portion_size;
    delete[] recv_portion_index;

    MPI_Finalize();

    return 0;
}
```

Компиляция приведенного выше кода осуществляется командой:

```
mpicc -O2 -openmp -mmic main.cpp -o
./lab1_dot_native_mpi.mic
```

Отличие от предыдущего примера заключает в использовании утилиты **mpicc** для компиляции кода с подключением заголовочных файлов и библиотеки MPI. Напомним, что для компиляции должна использоваться связка компилятора Intel и библиотеки Intel MPI.

Для запуска на четырех сопроцессорах Intel Xeon Phi с использованием MPI Hydra нужно выполнить команду:

```
mpiexec.hydra -hosts 4 mic0 mic1 mic2 mic3 -n 4 -perhost 1
./lab1_dot_native_mpi.mic
```

При работе на кластере с системой управления SLURM можно использовать следующую команду:

```
sbatch -N 2 --gres=mic:2 native_run.sh
./lab1_dot_native_mpi
```

Запуск будет выполнен на двух узлах, на каждом будет запущено по 2 процесса (в силу наличия на каждом узле по 2 Intel Xeon Phi).

Как и в предыдущем примере, для работы скрипта **native_run.sh** требуется подгрузить модуль **launcher/mic**, если это не было сделано ранее:

```
module load launcher/mic
```

Посмотреть статистику по количеству свободных и используемых узлов кластера можно командой:

```
sinfo
```

Результаты работы программы приведены на Рис. 6. Как и в предыдущем примере, результаты запишутся в файл **slurm-<номер задачи>.out**.

```
-sh-4.1$ cat slurm-9377.out
List of nodes:
node60-mic0
node61-mic0
node60-mic1
node61-mic1
Results:
128.796753
126.939026
122.497299
128.270416
120.303009
129.170853
113.081268
128.496140
118.298569
119.484879
126.637024
121.023445
123.013031
134.524841
125.303116
114.874908
117.644920
122.401176
125.044220
115.545937
```

Рис. 6. Результаты работы программы lab1_dot_native_mpi

2.3. Симметричный режим

В симметричном режиме в расчетах могут использоваться как центральные процессоры, так и сопроцессоры, причем каждое устройство является отдельным узлом кластера (обладает собственным уникальным MPI рангом). Обмен данными между устройствами осуществляется посредством передачи MPI сообщений.

Рассмотрим процесс компиляции и запуска программы в этом режиме. Воспользуемся предыдущим примером. Напомним, что задача состоит в параллельном вычислении набора скалярных произведений, причем каждое скалярное произведение так же считается параллельно в рамках одного процесса. Ранее мы использовали схему распараллеливания по процессам,

где каждый процесс выполнялся на Intel Xeon Phi и вычислял несколько скалярных произведений.

Можно отметить, что при таком подходе имеющиеся в системе центральные процессоры остаются не задействованными, хотя с успехом могут выполнять нашу программу без внесения в код каких-либо изменений. Задействовать все доступные вычислительные ресурсы как раз помогает симметричная модель исполнения. Основная идея ее использования состоит в том, чтобы запускать нашу программу одновременно на сопроцессорах и CPU.

Для этого, как уже отмечалось, менять код не нужно. Однако придется внести изменения в процессы компиляции и запуска.

В частности компилировать код нужно уже не только под сопроцессор, но и под CPU. Для этого нужно дополнительно выполнить команду компиляции для центрального процессора. Таким образом, имеем:

```
mpicc –O2 –openmp main.cpp –o ./lab1_dot_symmetric
mpicc –O2 –openmp –mmic main.cpp –o
./lab1_dot_symmetric.mic
```

В результате получим два отдельных исполняемых модуля.

Для запуска обоих модулей в симметричном режиме необходимо выполнить команду:

```
mpiexec.hydra –hosts 2 node0 node1 –n 2 –perhost 1
./lab1_dot_symmetric: \
–hosts 4 mic0 mic1 mic2 mic3 –n 4 –perhost 1 –wdir /tmp
/tmp/lab1_dot_symmetric.mic
```

Исполняемые файлы должны располагаться в каталоге **/tmp**. При этом наша программа будет запущена на 2 центральных процессорах и 4 сопроцессорах, т.е. всего на 6 узлах.

Для кластера с системой управления SLURM команда запуска будет такая:

```
export PPN=2
export MICperNODE=2
sbatch –N 2 --gres=mic:2 symmetric_run.sh
./lab1_dot_symmetric
```

Здесь программа будет запущена на двух узлах, на каждом будет выполнено 2 CPU и 2 MIC процесса. Всего 8 MPI процессов – 4 на центральных процессорах, и 4 на сопроцессорах. Обратите внимание, что для задания числа CPU процессов на узел мы используем переменную окружения **PPN**, число MIC процессов на узел равно **MICperNODE**. Имена переменных окружения могут быть разными для разных кластеров, и описаны обычно в руководстве пользователя для кластера.

Заметим, что исполняемый файл для сопроцессора в данном случае обязательно должен называться **lab1_dot_symmetric.mic** (т.е. к имени программы для центрального процессора добавляется расширение **.mic**).

Для работы этой команды требуется подгрузить модуль launcher/mic командой:

```
module load launcher/mic
```

```
-sh-4.1$ cat slurm-9390.out
List of nodes:
Node node195
Node node195
Node node196
Node node196
Node node195-mic1
Node node195-mic0
Node node196-mic1
Node node196-mic0
Results:
128.796753
126.939026
122.497299
128.270416
120.303001
129.170837
113.081268
128.496140
118.298569
119.484879
126.637024
121.023445
123.013031
134.524841
125.303116
114.874908
117.644920
122.401176
125.044220
115.545937
```

Рис. 7. Результаты работы программы lab1_dot_symmetric

3. Дополнительные задания

1. Реализовать умножение матрицы на вектор в режиме Offload.
2. Реализовать умножение матрицы на вектор в режиме работы только на сопроцессоре. Считать, что программа предназначена для выполнения на одном сопроцессоре.
3. Реализовать умножение матрицы на вектор в симметричном режиме. Обеспечить два уровня параллелизма – одновременное выполнение умножения строк матрицы на вектор и параллельное вычисление скалярного произведения векторов.

4. Литература

4.1. Использованные источники информации

1. Intel Xeon Phi Coprocessor System Software Developers Guide, revision 2.03, 2012
2. Best Known Methods for Using OpenMP on Intel Many Integrated Core (Intel MIC) Architecture, Volume 1a, January 29, 2013
3. Green R.W. Effective Use of the Intel Compiler's Offload Features: [http://software.intel.com/en-us/articles/effective-use-of-the-intel-compilers-offload-features]