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 №3
Оптимизация вычислений в задаче о разложении чисел на простые сомножители. Векторизация и балансировка нагрузки

Козинов Е.А.

При поддержке компании Intel

Нижний Новгород
2013

Содержание

Введение

Повышение производительности программ всегда была и будет актуальной проблемой. Для повышения производительности, эффективнее всего использовать алгоритмическую оптимизацию. К сожалению, алгоритмическая оптимизация, как правило, является очень трудоемкой. Второй путь повышения производительности программ – распараллеливание. В последнее время для распараллеливания используются не только ядра центрального процессора, но и различные ускорители и сопроцессоры. Одним из таких сопроцессоров является Intel Xeon Phi недавно появившейся на рынке.

В данной лабораторной работе рассматривается вопрос переноса вычислений с центрального процессора на сопроцессор Intel Xeon Phi на примере простой задачи – разложения чисел на простые сомножители. Также, рассматривается ряд простых подходов позволяющих увеличить производительность параллельной реализации алгоритма работающей на Intel Xeon Phi.

1. Методические указания

1.1. Цели и задачи работы

Цель данной лабораторной работы – рассмотрение на примере задачи разложения чисел на простые сомножители некоторых вопросов, возникающих при распараллеливании программ на сопроцессорах Intel Xeon Phi.Основные вопросы – метод переноса вычислений на сопроцессор, влияние разных методов распределения вычислений на скорость вычислений, а также подходы к оптиизации вычислений под сопроцессор Intel Xeon Phi.

Данная цель предполагает решение следующих основных задач:

1. Изучение простейшего алгоритма разложения чисел на простые сомножители[1].
2. Последовательная реализация алгоритма.

[1] Рассматриваемый алгоритм является малоэффективным, его использование в лабораторной работе носит иллюстративный характер. При решении практических задач, как правило, применяются другие алгоритмы, например, алгоритмы Полларда или Диксона ([1-3]).

3. Перенос вычислений на сопроцессор Intel Xeon Phi.
4. Многопоточная реализация данного алгоритма, которая предусматривает разделение множества факторизуемых чисел на равные части по количеству потоков.
5. Поиск других подходов к распределению нагрузки между потоками для повышения эффективности приложения.
6. Рассмотрение вопросов оптимизации программ под сопроцессор Intel Xeon Phi.
7. Построение гибридной схемы вычислений одновременно на CPU и сопроцессоре.

1.2. Структура работы

В начале, рассматривается задача разложения множества чисел на простые множители и приводится последовательная реализация. Далее рассматриваются вопросы переноса вычислений на сопроцессор Intel Xeon Phi и возможные проблемы, которые могут возникнуть в процессе переноса. В процессе выполнения лабораторной работы реализуется несколько многопоточных версий алгоритма в соответствии с различными подходами к распределению нагрузки между потоками. Для распараллеливания алгоритма применяются директивы OpenMP. Затем вычислительная схема оптимизируется за счет векторизации кода. В финале демонстрируется способ построения гибридной схемы вычислений одновременно на центральном процессоре и сопроцессоре.

1.3. Тестовая инфраструктура

Для проведения экспериментов использовались вычислительные ресурсы МСЦ РАН [4]. На момент проведения экспериментов тестовая инфраструктура представлена в таблице 1.

Таблица 1. Тестовая инфраструктура

Процессор	2 процессора на узел Xeon E5-2690 (2.9 GHz, 8 ядер)
Память	64 GB
Сопроцессор	Intel Xeon Phi 7110X
Операционная система	Linux CentOS 6.2
Компилятор, профилировщик, отладчик	Intel C/C++ Compiler 14

2. Задача разложения чисел на простые сомножители

Лабораторная работа построена на задаче факторизации (разложения на простые сомножители) чисел из диапазона от 1 до *N*. Используемый алгоритм базируется на попытке деления факторизуемого числа на каждое из меньших его чисел [2]. Если остаток от деления на некоторый множитель равен нулю, то этот множитель запоминается, частное становится делимым, после чего производится повторная попытка деления на это же число. Алгоритм завершает свою работу, когда частное от очередного деления равно единице.

Рассмотрим работу алгоритма для одного числа. Допустим, что необходимо определить простые множители числа 12. Для этого перебираются числа, меньшие 12, начиная с 2, и выполняется последовательность операций деления:

```
12 / 2 = 6  // остаток равен 0, пробуем делить еще раз на 2
6 / 2 = 3   // остаток равен 0, пробуем делить еще раз на 2
3 / 2 = 1,5 // остаток отличен от 0,
            //рассматриваем следующее число, меньшее 12
3 / 3 = 1   // частное равно 1, алгоритм останавливает работу
```

Псевдокод последовательного алгоритма выглядит следующим образом:

```
1. for i = 1 to N
2.   number ← i;
3.   for j = 2 to i
4.     if (number == 1) break;
5.     r ← number % j;
6.     if (r == 0)
7.       number ← number / j;
8.       save_divisor(i, j);
9.       j ← j - 1;
```

В соответствии с приведенным псевдокодом может быть реализована простейшая последовательная версия алгоритма.

3. Последовательная реализация алгоритма

Начнем реализацию алгоритма с создания файла `single.cpp`. Для этого можно выполнить следующую команду:

```
-sh-4.1$ >single.cpp
```

Далее созданный файл можно редактировать любым привычным редактором. Один из редакторов с дружественным интерфейсом `mcedit`. Для редактирования файла необходимо выполнить команду:

```
-sh-4.1$ mcedit ./single.cpp
```

В результате откроется окно редактора, в котором можно разрабатывать программную реализацию алгоритма.

Начнем разработку программы с подключения необходимых заголовочных файлов и объявления констант.

```
#include <iostream>
#include "omp.h"

#include <vector>

using namespace std;

// Количество факторизуемых чисел
#define NUM_NUMBERS 100000
// Вектора используемые для хранения
// простых сомножителей чисел
vector<int> divisors[NUM_NUMBERS+1];
```

Далее объявим две вспомогательные функции, используемые при реализации алгоритма:

```
// Получение количества создаваемых потоков
int testThreadCount();
// Функция факторизации чисел
void factorization();
```

Реализация данных функций будет приведена позднее.

Далее разработаем главную функцию программы `main`.

```
int main()
{
  // Обявление переменных
  double time_s, time_f;
  int intel_th;

  // Вывод количества создаваемых потоков
  intel_th = testThreadCount();
  cout << "Intel CPU thread:\n" << intel_th << endl;

  // Проведение вычислительного эксперимента
  time_s = omp_get_wtime( );
  factorization();
  time_f = omp_get_wtime( );

  cout<< "Calculation time : " << (time_f - time_s)
      << endl;

  // Вывод простых множителей произвольных 10 чисел
  for (int i = 0; i < 10; i++)
```

```
  {
    int randomIdx = 1 + rand() % NUM_NUMBERS;
    cout << randomIdx << ":\t";
    int size;
    size = static_cast<int>(divisors[randomIdx].size());
    for (int j = 0; j < size;j++)
    {
      cout << divisors[randomIdx][j] << "\t";
    }
    cout << endl;
  }
  return 0;
}
```

Ниже приведем код получения количества создаваемых потоков.

```
int testThreadCount() {
  int thread_count;
  #pragma omp parallel
  {
    #pragma omp single
      thread_count = omp_get_num_threads();
  }
  return thread_count;
}
```

В финале разработаем код алгоритма факторизации чисел приведенный в виде псевдокода разделе 2.

```
void factorization()
{
  for (int i = 1; i < NUM_NUMBERS; i++)
  {
    int number = i;
    int idx = number;

    for (int j = 2; j < idx; j++)
    {
      if (number == 1) break;

      int r;
      r = number % j;
      if (r == 0)
      {
        number /= j;
        divisors[idx].push_back(j);
        j--;
      }
    }
  }
}
```

Код готов.

Для проведения экспериментов в рамках вычислительных ресурсов МСЦ РАН необходимо вначале зарезервировать вычислительный узел. Сделать это можно с помощью команды `salloc`. Пример резервирования узла кластера представлен ниже:

```
-sh-4.1$ salloc -N 1 --gres=mic:1
salloc: Pending job allocation 9404
salloc: job 9404 queued and waiting for resources
salloc: job 9404 has been allocated resources
salloc: Granted job allocation 9404
```

Посмотреть имя выделенного хоста можно в переменной окружения `SLURM_NODELIST`.

```
-sh-4.1$  echo $SLURM NODELIST
node196
```

Перейдем на выделенный хост.

```
-sh-4.1$ ssh $SLURM_NODELIST
-sh-4.1$ hostname
node196
```

Далее откомпилируем код и выполним его.

```
-sh-4.1$ icpc -O2 -openmp single.cpp -osingle
-sh-4.1$ ./single
```

На Рис 1. приведен результат работы последовательной версии алгоритма.

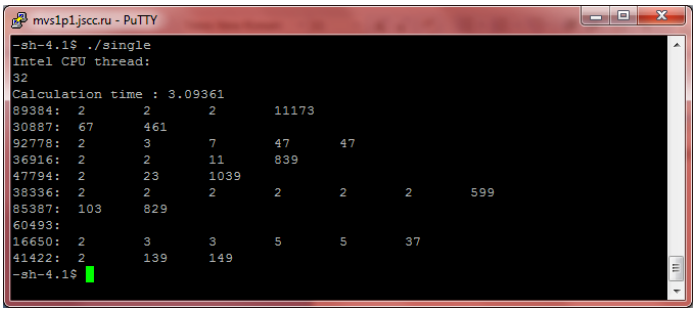

Рис 1. Последовательный алгоритм факторизации чисел

Ниже, на Рис 2. приведено время работы алгоритма:

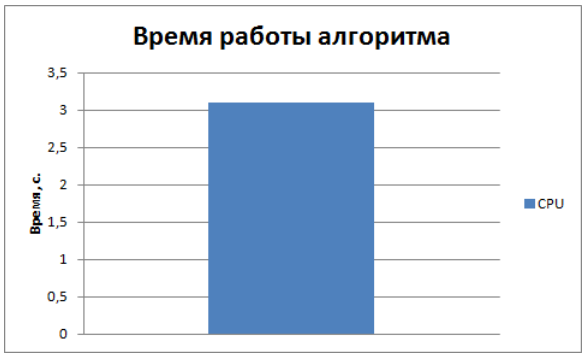

Рис 2. Время работы последовательной факторизации на CPU

4. Перенос вычислений на сопроцессор Intel Xeon Phi

В качестве следующего шага перенесем вычисления последовательного кода на сопроцессор Intel Xeon Phi. Для этого воспользуемся директивой `#pragma offload target(mic:0)`.

```
  time_s = omp_get_wtime( );
#pragma offload target(mic:0)
  {
  factorization();
  }
  time_f = omp_get_wtime( );
```

Далее проведем попытку скомпилировать полученный код. Получим следующий вывод компилятора с ошибками:

```
./singleMIC.cpp(44): error: function "factorization" called
in offload region must have been declared with compatible
"target" attribute
    factorization();
```

Intel Xeon Phi имеет свои операционную систему и окружение. Код, исполняемый на MIC, должен быть скомпилирован отдельно. В программной реализации алгоритма факторизации чисел не было указаний компилятору дополнительно скомпилировать для MIC функцию факторизации, что и привело к появлению ошибки. Чтобы исправить данную ошибку, необходимо объявление функций и сами функции обернуть директивами, как показано ниже:

```
#pragma offload_attribute(push, target(mic))
#include <iostream>
#include "omp.h"

#include <vector>
```

```
using namespace std;


#define NUM_NUMBERS 100000
vector<int> divisors[NUM_NUMBERS+1];


void factorization(int chunk);
int  testThreadCount();
#pragma offload_attribute(pop)

…

#pragma offload_attribute(push, target(mic))
int testThreadCount() {
  int thread_count;
  #pragma omp parallel
  {
    #pragma omp single
      thread_count = omp_get_num_threads();
  }
  return thread_count;
}

void factorization()
{…}
#pragma offload_attribute(pop)
```

Попробуем скомпилировать программу. Программа должна откомпилироваться без ошибок.

Для того чтобы убедится, что вычисления производятся на сопроцессоре, и узнать его параметры (количество создаваемых потоков), дополнительно модифицируем функцию `main`.

```
int main()
{
  double time_s, time_f;
  int intel_th, mic_th;

  intel_th = testThreadCount();
  cout << "Intel CPU thread:\n" << intel_th << endl;

  int number_of_coprocessors =
_Offload_number_of_devices();

  if(number_of_coprocessors < 1)
  {
    cout << "for the program to need at least one Intel
Xeon Phi coprocessor \n";
    return -1;
```

```
  }
  cout << "Intel Xeon Phi:\n";
  cout << "Number of coprocessors: " <<
number_of_coprocessors << endl;
  cout << "(used only the first coprocessor)" << endl;

#pragma offload target(mic:0)
  {
    mic_th = testThreadCount();
  }
  cout << "MIC thread:\n" << mic_th << endl;

  time_s = omp_get_wtime( );
#pragma offload target(mic:0)
  {
  factorization();
  }
  time_f = omp_get_wtime( );
  …
}
```

Откомпилируем и запустим программу. Результат выполнения программы представлен на Рис 3.

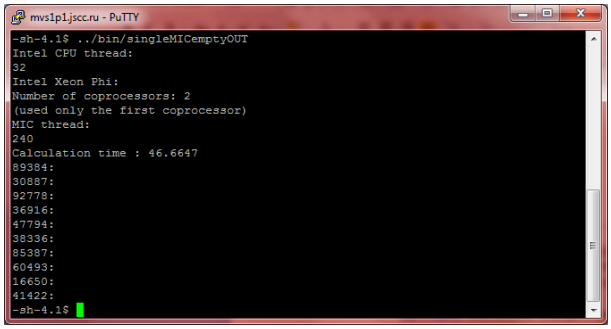

Рис 3. Результат вычислений простых чисел с перенесенными вычислениями на сопроцессор MIC

Как вы думаете, почему в результате вычислений не отобразились факторы чисел?

Правильный ответ – *память центрального процессора и сопроцессора не являются общими.* Как следствие при объявлении глобальных данных, память под хранения данных выделяется как на центральном процессоре, так и на сопроцессоре. Изменение данных на сопроцессоре не влечет изменение данных на центральном процессоре.

Разработаем дополнительно две функции, которые позволят получить данные с сопроцессора.

Первая функция получает на вход количество изымаемых чисел и их номера. Последним параметром передается массив размера на единицу больше количества получаемых чисел. Каждых элемент массива – сумма размеров длин векторов для изымаемых факторов чисел начиная с первого и до текущего. Последний элемент массива – общее количество чисел фактора, которое необходимо получить на центральном процессоре. Ниже приведен код функции:

```
void getSizeVector(int count, int * num, int *size)
{
  int sum = 0;
  int i = 0;
  for (i = 0; i < count; i++)
  {
    int Idx = num[i];
    size[i] = sum ;
    sum += static_cast<int>(divisors[Idx].size());
  }
  size[i] = sum;
}
```

Вторая функция также получает на вход количество изымаемых чисел и их номера. Возвращает функция последовательно записанные в вектор факторы чисел. Ниже приведен код второй функции:

```
void getVector(int count, int * num, int *pn)
{
  int i = 0;
  int size = 0;
  int k = 0;
  for (i = 0; i < count; i++)
  {
    int Idx = num[i];
    size = static_cast<int>(divisors[Idx].size());
    for (int j = 0; j < size;j++)
    {
      pn[k] = divisors[Idx][j];
      k++;
    }
  }
}
```

Модифицируем код функции `main` так, чтобы факторы чисел выводились корректно.

```
int main()
{
…
  time_s = omp_get_wtime( );
  #pragma offload target(mic:0)
  {
```

```
  factorization();
  }
  time_f = omp_get_wtime( );

  cout<< "Calculation time : " << (time_f - time_s)
  << endl;

  // Получаем количество простых сомножителей
  int num[10], countNum[11], * pn, sum;
  for (int i = 0; i < 10; i++)
  {
    int randomIdx = 1 + rand() % NUM_NUMBERS;
    num[i] = randomIdx;
  }

  #pragma offload target(mic:0)
          in(num[0:10]) out(countNum[0:11])
  {
  getSizeVector(10, num, countNum);
  }
  sum = countNum[10];
  pn  = new int [sum];

  #pragma offload target(mic:0)
          in(num[0:10]) out(pn[0:sum])
  {
  getVector(10, num, pn);
  }
  // Вывод простых сомножителей произвольных 10 чисел
  for (int i = 0; i < 10; i++)
  {
    cout << num[i] << ":\t";
    for (int j = countNum[i]; j < countNum[i + 1];j++)
    {
      cout << pn[j] << "\t";
    }
    cout << endl;
  }
  delete []pn;

  return 0;
}
```

Скомпилируем и выполним полученный код. На Рис 4. приведен результат выполнения программы.

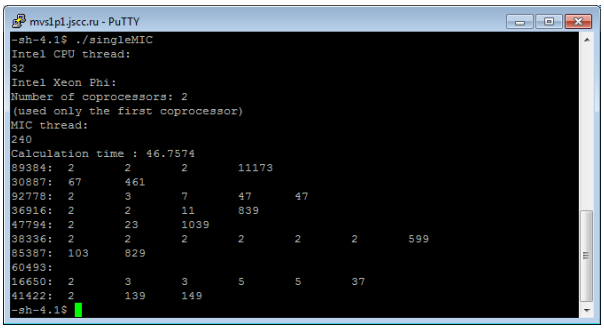

Рис 4. Результат вычислений простых чисел с перенесенными вычислениями на сопроцессор MIC

На Рис 5. приведен график сравнения производительности кода, запущенного на центральном процессоре и на Intel Xeon Phi.

Рис 5. Сравнение времени вычислений простых чисел на CPU и MIC

Как видно из графика, центральный процессор приблизительно в пятнадцать раз обогнал сопроцессор Intel Xeon Phi. Причина заключается в том, что ядра сопроцессора гораздо более простые (с точки зрения архитектуры) и обладают меньшей тактовой частотой, чем ядра центрального процессора. Основное преимущество Intel Xeon Phi в том, что ядер, пусть и простых, очень много. У сопроцессора Intel Xeon Phi 60 ядер против 8 центрального процессора. Также каждое ядро Intel Xeon Phi поддерживает четыре потока. Итого на ускорителе может выполняться 240 потоков. Следует учитывать, что для эффективного использования ядер сопроцессора необходимо, чтобы код был векторизован.

5. Параллельная реализация алгоритма

Для реализации параллельного алгоритма воспользуемся директивами OpenMP. Вначале произведем распараллеливание, используя статический

планировщик. Затем, попробуем увеличить производительность за счет изменения планировщика.

5.1. Распределение вычислений с использованием статического планировщика

Простейший подход к распараллеливанию в задаче разложения чисел из диапазона от 1 до *N* состоит в том, чтобы разделить множество факторизуемых чисел на равные части по числу потоков. На Рис 6. показан пример распределения чисел при создании четырех потоков.

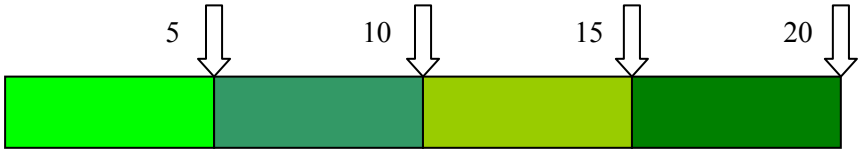

Рис 6. Распределение нагрузки между потоками – вариант 1

В случае применения директив OpenMP подобного распределения можно добиться, используя статический планировщик, используемый по умолчанию.

Модифицируем код факторизации с учетом выбранного способа распараллеливания.

```
void factorization()
{
#pragma omp parallel for
  for (int i = 1; i < NUM NUMBERS; i++)
  {
    int number = i;
    int idx = number;

    for (int j = 2; j < idx; j++)
    {
      if (number == 1) break;

      int r;
      r = number % j;
      if (r == 0)
      {
        number /= j;
        divisors[idx].push_back(j);
        j--;
      }
    }
  }
}
```

Скомпилируем и выполним код. На Рис 7. приведен результат вычислений параллельного кода исполненного на ускорителе Intel Xeon Phi с применение статического планирования.

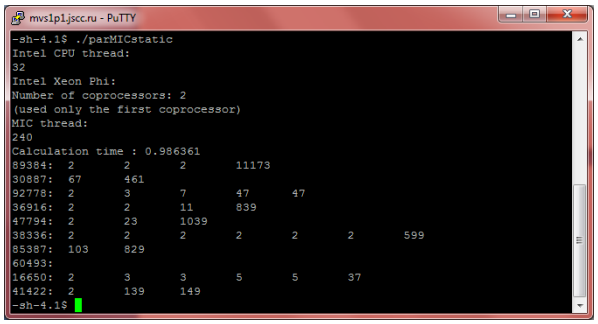

Рис 7. Результат вычислений параллельного кода на MIC с применение статического планирования

На Рис 8. представлен график демонстрирующий ускорение параллельной реализации алгоритма.

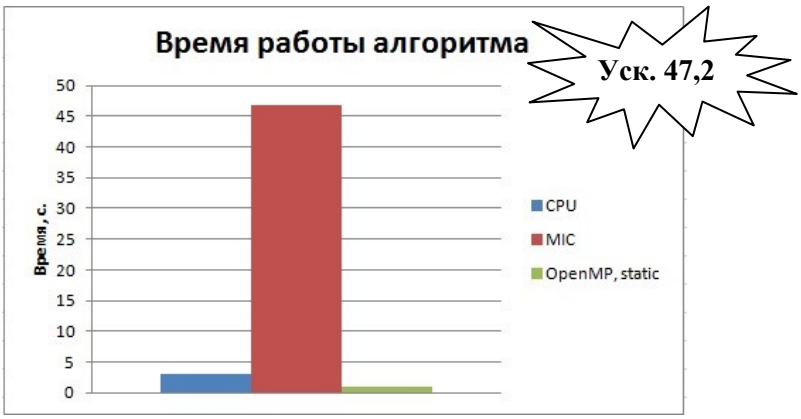

Рис 8. Сравнение времени вычислений последовательных и параллельной реализаций

Ускорение алгоритма факторизации чисел относительно последовательной версии исполненных на Intel Xeon Phi составило 47,2 раза. Результат можно считать неплохим, но далеким от идеала. Шестьдесят ядер могли обеспечить гораздо лучшее ускорение.

Что помешало большему ускорению?

Ответ простой, вычисления сильно разбалансированы и большое время тратится на синхронизацию потоков. Фактор для чисел первых потоков можно найти гораздо быстрее, чем для последних чисел.

5.2. Распределение вычислений с использованием динамического планировщика

Попробуем сбалансировать нагрузку. Для этого воспользуемся дополнительными возможностями директивы `#pragma omp parallel for` – изменим планировщик со статического на динамический. При динамическом планировании числа будут чередоваться между потоками. Если потока четыре, то первое число достанется первому потоку, второе – второму и т.д. Следующее число, которое получит первый поток, будет 5. Второй 6 и т.д. На Рис 9. показан пример распределения чисел при создании четырех потоков.

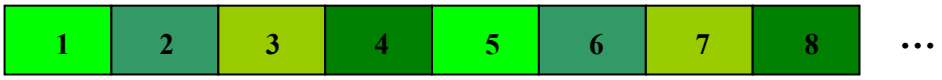

Рис 9. Распределение нагрузки между потоками – вариант 2

Ниже приведен код факторизации чисел с использованием динамического планирования.

```
void factorization()
{
#pragma omp parallel for schedule(dynamic)
  for (int i = 1; i < NUM_NUMBERS; i++)
  {
    int number = i;
    int idx = number;

    for (int j = 2; j < idx; j++)
    {
      if (number == 1) break;

      int r;
      r = number % j;
      if (r == 0)
      {
        number /= j;
        divisors[idx].push_back(j);
        j--;
      }
    }
  }
}
```

Скомпилируем и выполним код. На Рис 10. представлен результат выполнения программы при использовании динамического планирования.

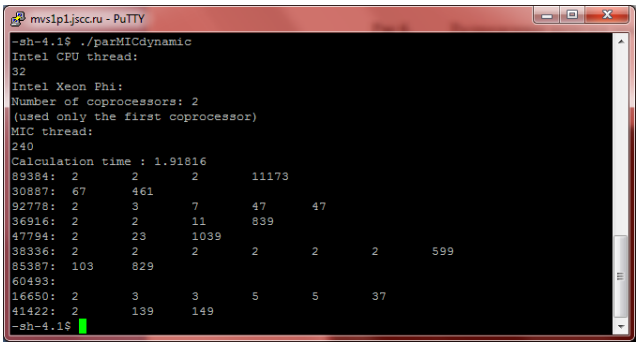

Рис 10. Результат вычислений параллельного кода на MIC с применение статического планирования.

На рис. 11 приведен график сравнения времени выполнения параллельных алгоритмов со статическим и динамическим планированием.

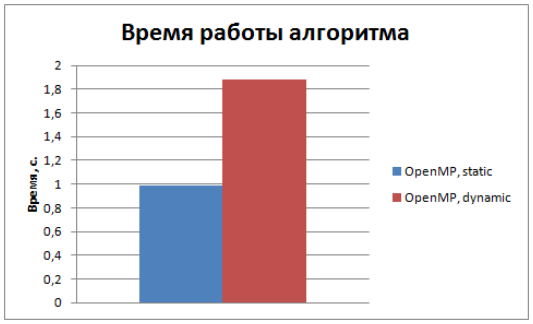

Рис 11. График сравнения времени выполнения параллельных алгоритмов со статическим и динамическим планированием.

Как видно из графика время выполнения алгоритма замедлилось почти в два раза.

В чем причина подобного замедления?

Основная причина в методе распределения чисел. Все четные потоки получают четные числа. Потоков четное число. Как следствие четные ядра сопроцессора будут выполнять работы вдвое меньше. Взамен того, чтобы сбалансировать вычисления мы добились большего дисбаланса. Так как потоков много, дисбаланс при статическом планировании оказался менее ярко выраженным.

5.3. Распределение вычислений с использованием динамического планировщика и подбор размера порции

Рассмотрим еще один подход к распределению нагрузки в задаче разложения чисел из диапазона от 1 до *N*. Суть подхода состоит в том, чтобы разделить множество факторизуемых чисел на группы и делить группы между

потоками чередованием. На Рис 12. показан пример распределения чисел при создании четырех потоков.

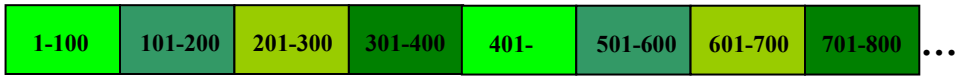

Рис 12. Распределение нагрузки между потоками – вариант 3

Модифицируем код функции факторизации так, чтобы можно было подобрать размер порции.

```
void factorization(int chunk)
{
#pragma omp parallel for schedule(dynamic, chunk)
  for (int i = 1; i < NUM_NUMBERS; i++)
  {
    int number = i;
    int idx = number;

    for (int j = 2; j < idx; j++)
    {
      if (number == 1) break;

      int r;
      r = number % j;
      if (r == 0)
      {
        number /= j;
        divisors[idx].push_back(j);
        j--;
      }
    }
  }
}
```

Дополнительно реализуем функцию очистки списков факторов чисел.

```
void clean()
{
  for (int i = 1; i < NUM_NUMBERS; i++)
  {
    divisors[i].resize(0);
  }
}
```

Модифицируем функцию `main` для поиска оптимального размера порции.

```
int main()
{
…
  cout << "Calculation time : " << endl;
  for (int i = 10; i < 100; i+=10)
```

```
  {
    #pragma offload target(mic:0)
    {
    clean();
    }
    chunk = i;
    time_s = omp_get_wtime( );
    #pragma offload target(mic:0)
    {
    factorization(chunk);
    }
    time_f = omp_get_wtime( );

    cout << chunk << ";" << (time_f - time_s) << endl;
  }

  for (int i = 0; i <= 10; i++)
  {
    #pragma offload target(mic:0)
    {
    clean();
    }
    chunk = 100 + i * 50;
    time_s = omp_get_wtime( );
    #pragma offload target(mic:0)
    {
    factorization(chunk);
    }
    time_f = omp_get_wtime( );

    cout << chunk << ";" << (time_f - time_s) << endl;
  }
…
}
```

Скомпилируем и выполним код. На Рис 13. представлен результат выполнения программы.

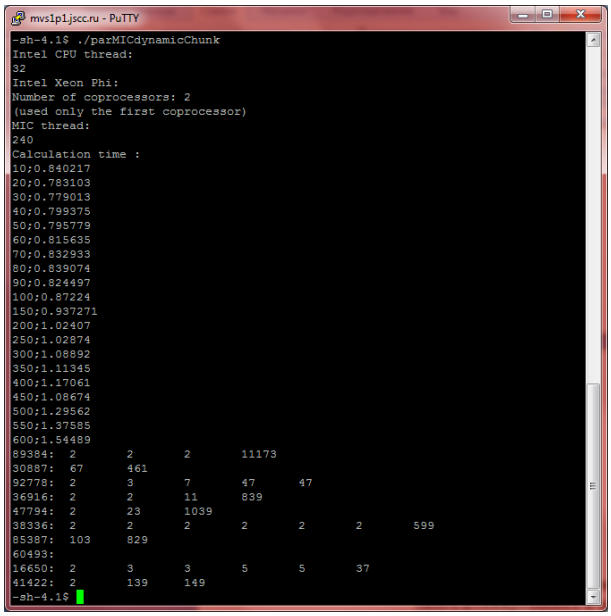

Рис 13. Подбор оптимального размера порции при динамическом планировании

На Рис 14. представлен график времени выполнения алгоритма факторизации чисел, при разных размерах порции.

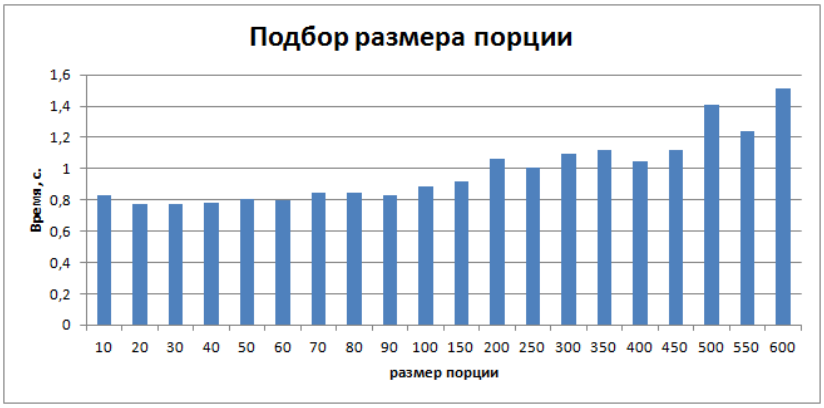

Рис 14. График времени выполнения алгоритма факторизации при разных размерах порции.

Из графика видно, что оптимальный размер порции составляет 30. На Рис 15. представлен график сравнения времени выполнения реализованных параллельных реализаций алгоритма факторизации чисел.

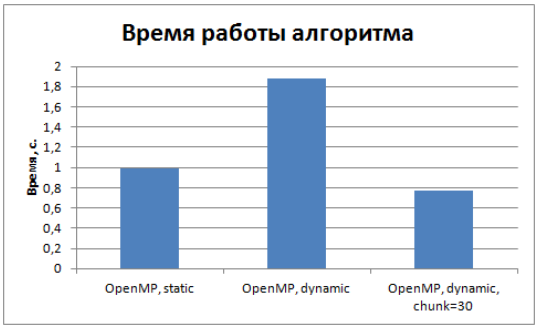

Рис 15. График сравнения времени выполнения параллельных реализаций.

Как видно из представленного графика, динамическое планирование при правильном подборе размера порции позволило получить большее ускорение вычислений.

Применение динамического планирования не является единственным рычагом повышения производительности кода. В дополнительных заданиях предлагается попробовать ряд других оптимизаций.

5.4. Оптимизация вычислительной функции

В предыдущих разделах лабораторной работы был рассмотрен наивный подход к переносу вычислений с центрального процессора на сопроцессор. Попробуем оценить полученный результат. Для этого сравним время работы параллельной версии алгоритма на центральном процессоре и сопроцессоре Intel Xeon Phi. Для этого временно закомментируем директивы отправки вычислений на сопроцессор и скомпилируем код.

Ниже представлен график сравнения времени вычислений параллельных реализаций алгоритма факторизации чисел на процессоре и на сопроцессоре, при размере порции в динамическом планировании равным 30.

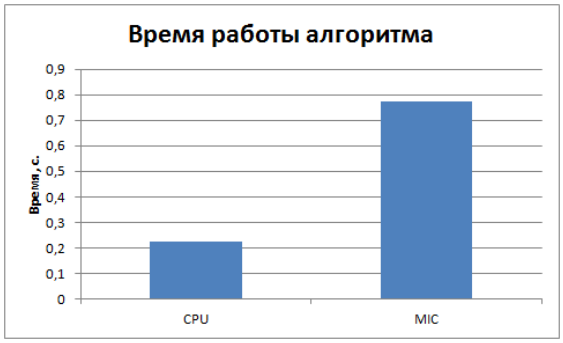

Рис 16. Сравнение времени вычислений на процессоре и сопроцессоре

Исходя из графика видно, что лучшее время параллельного алгоритма на сопроцессоре в почти четыре раза хуже, чем время на центральном процессоре. Вместе с тем, исходя из характеристик, сопроцессор Intel Xeon Phi обладает большей пиковой производительностью.

В нашей программе все вычисления независимы. *Почему не удается достичь большей производительности?*

Ответ на данный вопрос следующий. Для достижения пиковой производительности ускорителя необходимо задействовать не только все ядра процессора, но и код должен быть векторизуем. Векторные операции вносят существенный вклад в пиковой производительности сопроцессора Intel Xeon Phi.

Для того чтобы понять, векторизовался ли код, необходимо собрать отчет о векторизации.

```
icpc -O2 -openmp -vec-report3  parMIC.cpp
```

Посмотрим полученный отчет:

```
…
parMIC.cpp(174): (col. 5) remark: *MIC* loop was not vec-
torized: unsupported loop structure
parMIC.cpp(169): (col. 3) remark: *MIC* loop was not vec-
torized: nonstandard loop is not a vectorization candidate
…
```

Строки 169 и 174 соответствуют циклам функции факторизации. Циклы не векторизовались.

Intel Xeon Phi содержит регистры длиной 512 бит для векторных операций. В разрабатываемой программе используются целые числа типа int. Размер int 4 байта. Как следствие если бы код векторизовался, за раз можно было бы выполнить 16 операций.

Рассмотрим внимательно вычислительный цикл. В цикле все операции строго последовательные. Без внесения не тривиальных алгоритмических изменений его не векторизовать.

В рассматриваемой задаче основной операцией является поиск остатка от деления и сравнение с 0. Фактор имеет несравнимо малое число делителей по сравнению с количеством делений. Как правило, число не делится. Данных факт можно попробовать использовать для ускорения вычислений. Можно завести массив из 16 чисел (размер регистра). В каждый элемент массива можно вычислить остаток от деления от текущего делителя плюс индекс элемента массива. Полученные делители можно перемножить. Если результат умножения не равен нулю, то 16 чисел можно уже не проверять на делимость. В противном случае необходимо выполнить исходную проверку на делимость и сместить «окно» проверяемых делителей.

Циклы нахождения остатка и произведения векторизуемы. Ниже представлена программная реализация модификации алгоритма.

```
#define LOOP_SIZE 16
…
void factorization(int chunk)
{

#pragma omp parallel for schedule(dynamic, chunk)
  for (int i = 1; i < NUM_NUMBERS; i++)
  {
    int rr[LOOP_SIZE];
    int r, p;
    int number = i;
    int idx = number;

    for (int j = 2; j < idx; j++)
    {
      if (number == 1) break;
        #pragma simd
        for(int k = 0; k < LOOP_SIZE; k++)
        {
            rr[k] = number % (j + k);
        }
        p = 1;
        #pragma simd
        for(int k = 0; k < LOOP_SIZE; k++)
        {
            p *= rr[k];
        }
        if(p != 0)
        {
            j += LOOP_SIZE - 1;
        } else
        {
              r = number % j;
              if (r == 0)
              {
                  number /= j;
                  divisors[idx].push_back(j);
                  j--;
              }
        }
    }
  }
}
```

Откомпилируем и запустим код:

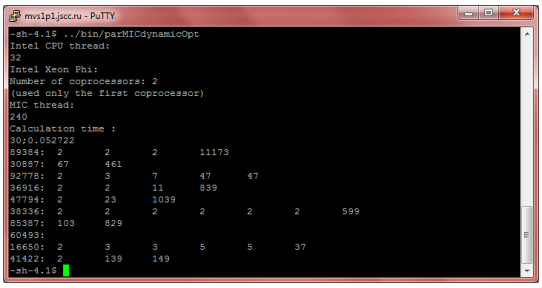

Рис 17. Результат вычислений с оптимизированной версией кода

Ниже представлен график сравнения времени вычислений на сопроцессоре Intel Xeon Phi оптимизированной и не оптимизированной версий кода.

Рис 18. Сравнение оптимизированной и не оптимизированной программной реализации на Intel Xeon Phi

Особенностью оптимизации под Intel Xeon Phi является то, что оптимизация также положительно влияет и на время вычислений на центральном процессоре. Ниже представлен график сравнения времени вычислений на центральном процессоре.

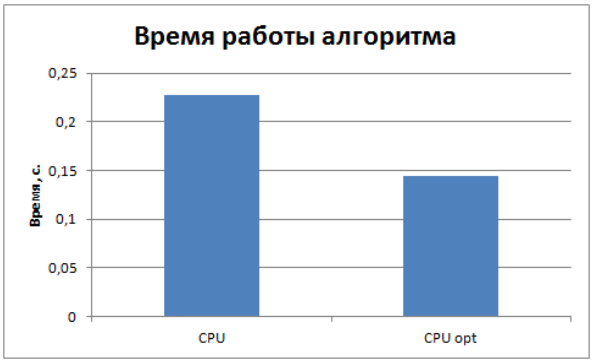

Рис 19. Сравнение оптимизированной и не оптимизированной программной реализации на CPU

Из графика видно, что за счет векторизации кода, на центральном процессоре время уменьшилось в полтора раза.

Ниже приведен график сравнение времени работы оптимизированных версий факторизации на CPU и сопроцессоры.

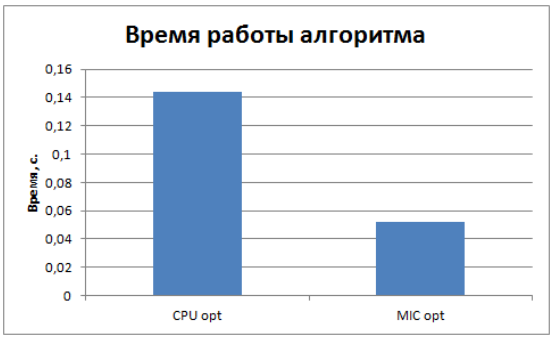

Рис 20. Сравнение времени вычислений на процессоре и сопроцессоре

Из экспериментов видно, что оптимизированная реализация алгоритма факторизации на Intel Xeon Phi смогла превзойти оптимизированную версию на CPU почти в три раза.

5.5. Гибридная параллельная схема реализации алгоритма

Последним шагом попробуем организовать схему вычислений одновременно на центральном процессоре и на сопроцессоре, что бы задействовать все имеющиеся ресурсы.

Так как времена стали слишком маленькие, чтобы была возможность сделать какие-то выводы, увеличим количество факторизуемых чисел в 10 раз и замерим время работы алгоритма. Ниже представлены времена работы алгоритмов на сопроцессоре и CPU.

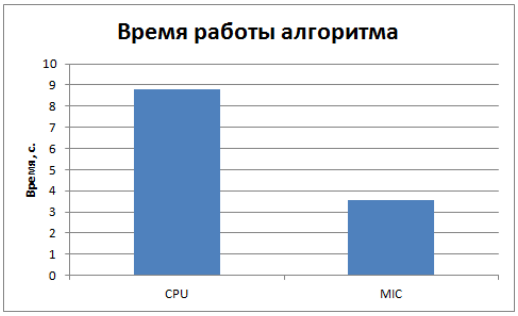

Рис 21. Время факторизации при увеличении количества чисел в 10 раз

Из графика видно, что соотношение времен сохранилось.

Директива `#pragma offload target(mic:0)` по умолчанию является синхронной. В данной директиве есть дополнительная опция позволяющая выполнить вычисления асинхронно. Для этого в директиве объявляется сигнал. Для синхронизации сигнал можно проверять и ожидать. Используя сигналы, осуществим статическое распределение нагрузки между процессором и сопроцессором. Для этого, в начале, отправим часть вычислений на Intel Xeon Phi в асинхронном режиме. Оставшуюся часть чисел факторизуем на центральном процессоре.

Модифицируем код основной функции следующим образом:

```
  float *f1;
  chunk = NUM_NUMBERS / 2;
  time_s = omp_get_wtime( );
  #pragma offload target(mic:0)  signal(f1)
  {
  factorization(chunk, 1);
  }
  factorization(NUM_NUMBERS - chunk, chunk + 1);
  #pragma offload target(mic:0) wait(f1)
  {
  end();
  }

  time_f = omp_get_wtime( );
```

Также изменим функцию факторизации для управления распределениями вычислений:

```
void factorization(int chunk, int start)
{
#pragma omp parallel for schedule(dynamic, 30)
  for (int i = start; i < start + chunk; i++)
  {
…
    }
  }
}
```

Функция end ничего не делает и используется как барьер синхронизации.

Код, связанный с выводом простых чисел, предлагается поправить самостоятельно.

Попробуем запустить код. Ниже представлено сравнение времени вычислений.

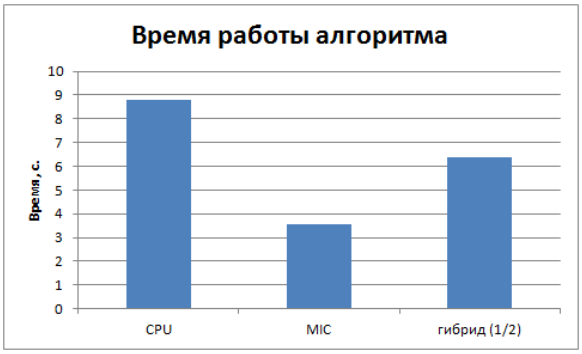

Рис 22. Разделение чисел поровну

Как видно из графика мы получили время вычислений среднее между временем сопроцессора и процессора.

С чем это связано?

Это связано с некорректным распределением нагрузки. Во-первых, время факторизации маленьких и больших чисел не одинаково. Во-вторых, процессор и сопроцессор обладают разной вычислительной производительностью. Попробуем сдвинуть границу. Первые 8/10 чисел отдадим сопроцессору, а оставшиеся вычислим на CPU.

Ниже представлен график сравнения времени вычислений с новым распределением нагрузки.

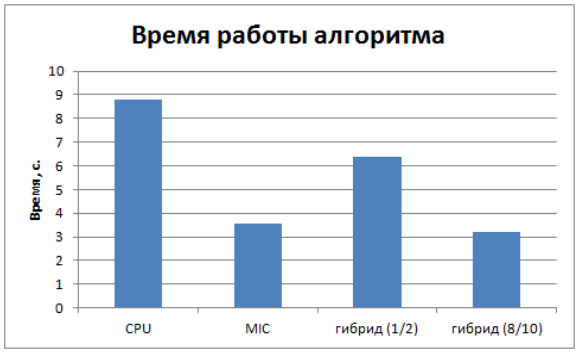

Рис 23. Разделение чисел 8/10

Как видно из графика, сдвиг границы порций вычислений позволил улучшить время вычислений в целом. В качестве дополнительной задачи предлагается более точно найти границу порций разделения вычислений между процессором и сопроцессором.

6. Дополнительные задания

1. Попробуйте путем изменения количества потоков OpenMP увеличить производительность параллельного алгоритма факторизации чисел. Учтите, что особенностью сопроцессора Intel Xeon Phi является то, что большая производительность может проявиться при наличии нескольких потоков OpenMP на ядро.

2. Проверьте, можно ли повысить производительность алгоритма факторизации чисел при нечетном размере порции.

3. Изучите, какие алгоритмические оптимизации можно выполнить, для повышения производительности кода.

4. Рассмотрите, какие программные оптимизации можно выполнить еще с кодом для повышения производительности.

5. Более точно подберите границу разделения порции вычислений для сопроцессора и процессора в гибридной схеме.

7. Литература

1. Василенко О.Н. Теоретико-числовые алгоритмы в криптографии. 2003. – М.: МЦНМО, 2003.Стр. 78-83.
2. Кнут Д. Искусство программирования. – М.: Вильямс, 2007. Т.2, с. 465-468.
3. Черемушкин А.В. Лекции по арифметическим алгоритмам в криптографии. – М.: МЦНМО, 2002. Стр. 77-80.
4. Международный Вычислительный Центр Российской Академии Наук. – aURL: http://www.jscc.ru/ (дата обращения: 10.09.2013)