Обучение основам разработки приложений для современных архитектур высокопроизводительных вычислительных систем

С.А.Немнюгин

Санкт-Петербургский государственный университет

Russian Supercomputing Days 2017

Суперкомпьютерное образование: проблемы и перспективы

Обучение основам разработки приложений для современных архитектур высокопроизводительных вычислительных систем.

Курсы направления «Прикладные физика и математика», Санкт-Петербургский государственный университет.

Бакалавриат.

1 курс. «Введение в современные компьютерные технологии» (1-2 семестры).
4 курс. «Практикум по параллельным и распределенным вычислениям» (8 семестр).

Магистратура.

1 курс. «Современные технологии программирования в научных исследованиях-I» (1 семестр).
«Современные технологии программирования в научных исследованиях-II» (2 семестр).
«Методы и технологии высокопроизводительных вычислений-I» (1 семестр).
«Методы и технологии высокопроизводительных вычислений-II» (2 семестр).

2 курс. «Основы параллельного программирования-I» (3 семестр).
«Основы параллельного программирования-II» (4 семестр).

«Современные технологии программирования в научных исследованиях-I и II»

Темы:

- Современные информационные технологии, архитектуры современных вычислительных систем и тенденции их развития.
- Модели вычислений, методы теоретического анализа алгоритмов, законы масштабируемости.
- Программные инструменты разработки и оптимизации высокопроизводительных приложений.
- Компиляторная оптимизация прикладного программного обеспечения.
- Средства и технологии параллельного программирования для вычислительных систем с общей памятью (многоядерных). OpenMP.
- Средства и технологии параллельного программирования для вычислительных систем с распределённой памятью. Message Passing Interface.
- Основы программирования для GPGPU (General Purpose Graphic Processor Unit). Идеология. CUDA, OpenCL.
- Средства и технологии параллельного программирования для вычислительных систем с общей памятью (многоядерных). Часть 2. Intel® Cilk™ Plus.
- Алгоритмы высокопроизводительных и параллельных вычислений.

Целью учебных занятий является изучение и получение навыков использования технологий программирования для современных архитектур вычислительных систем.

Научное программирование часто используется для решения трудоёмких задач, связанных с математическим моделированием, обработкой данных и т. д.

Успешное решение этих задач требует эффективного использования современных вычислительных систем. Их особенностью является многоуровневый параллелизм, гетерогенность – применение наряду с процессорами классического типа ускорителей вычислений.

Большое значение имеет также оптимизация научных приложений, обеспечение их корректной работы, повышение производительности.

Для достижения указанных целей в рамках данного учебного курса решаются следующие задачи:

- ✓Даётся обзор современных архитектур вычислительных систем с углублённым анализом их ключевых особенностей.
- ✓Даётся обзор современных технологий программирования, которые используются для разработки высокопроизводительных вычислительных приложений.
- ✓Рассматриваются средства динамического и статического анализа программ, средства прогнозирования их эффективности и соотнесения с теоретическими оценками производительности в духе модели Амдала.
- ✓Учащиеся получают навык практической работы с программными инструментами динамического и статического анализа.
- ✓Даётся обзор математических библиотек численных методов и базовых операций. Учащиеся получают навык использования математических библиотек, выполняют сравнительное исследование эффективности и масштабируемости приложений, в которых используются математические библиотеки.
- ✓Даётся обзор возможностей автоматической оптимизации программ при компиляции. Знакомство с этими возможностями выполняется и на практике.

Важнейшей задачей, решаемой в данном курсе, является знакомство с такими технологиями разработки высокопроизводительных приложений, как Pthreads, Windows API, OpenMP, MPI, Cilk Plus, CUDA и другие. Учащиеся знакомятся не только с перечисленными технологиями, но и особенностями их применения, достоинствами и ограничениями, «подводными камнями», которые могут встретиться в процессе программирования, средствами динамического анализа параллельных программ.

В рамках данного курса также рассматриваются фундаментальные законы Computer Science, играющие важнейшую роль в высокопроизводительных вычислениях, высокопроизводительные и параллельные алгоритмы решения типичных вычислительных задач.

Требования к подготовленности обучающегося к освоению содержания учебных занятий

✓Знания в объёме вводного курса по информатике и программным системам.
✓Базовые знания архитектуры вычислительных систем.
✓Навыки работы в операционных системах Windows/Linux на уровне «продвинутого» пользователя.
✓Базовые знания и навыки программирования на языках C/C++.

Перечень результатов обучения

- ✓знание современных архитектур вычислительных систем и понимание тенденций их развития;
- ✓знание современных технологий программирования, которые используются для разработки высокопроизводительных вычислительных приложений;
- ✓навыки использования средств динамического и статического анализа программ, средств прогнозирования их эффективности;
- ✓знание особенностей математических библиотек численных методов и базовых операций, и навыки их использования;
- ✓знание возможностей автоматической оптимизации программ при компиляции и навык их практического применения;
- ✓знание таких технологий разработки высокопроизводительных приложений, как, OpenMP, MPI, CUDA и других, навыки их применения;
- ✓знание фундаментальных законов Computer Science, играющих важнейшую роль в высокопроизводительных вычислениях;
- ✓знание высокопроизводительных и параллельных алгоритмов решения типичных вычислительных задач.

Трудоёмкость, объёмы учебной работы и наполняемость групп обучающихся																		
Период обучения (модуль)	Контактная работа обучающихся с преподавателем												Самостоятельная работа				Объём активных и интерактивных форм учебных занятий	Трудоёмкость
	лекции	семинары	консультации	практические занятия	лабораторные работы	контрольные работы	коллоквиумы	текущий контроль	промежуточная аттестация	итоговая аттестация	под руководством преподавателя	в присутствии преподавателя	сам.раб. с использованием методических материалов	текущий контроль (сам.раб.)	промежуточная аттестация (сам.раб.)	итоговая аттестация (сам.раб.)		
ОСНОВНАЯ ТРАЕКТОРИЯ																		
очная форма обучения																		
Семестр 1	30		2						2				45		29		4	3
	2-20		2-20						2-20				1-1		1-1			
ИТОГО	**30**		**2**						**2**				**45**		**29**			**3**

Особенность перечисленных курсов – фокус на отдельных темах/технологиях программирования, недостаточный акцент на взаимосвязь технологий и их эффективное совместное использование.

Проект создания курса «Разработка гибридного прикладного программного обеспечения»

- Причины разработки курса.
- Место курса в учебном процессе.
- Содержание курса.
- Аппаратное и программное обеспечение.

Проект направлен на разработку учебного курса для магистратуры, посвященного методам создания прикладного программного обеспечения для современных и перспективных компьютерных архитектур.

Такие архитектуры вычислительных систем как высокопроизводительных, так и класса автоматизированных рабочих мест исследователя и технического специалиста нередко являются ***гибридными***. Эффективное программирование для таких систем основано на разных подходах, использовании разного программного инструментария и разных алгоритмов. Целью данного проекта является разработка курса, который позволит дать специалисту-разработчику прикладного программного обеспечения необходимые компетенции в области эффективного использования гибридных вычислительных систем.

Гибридная вычислительная платформа

Определение 1. Гибридная вычислительная платформа – вычислительная система, в которой используются как «обычные» процессоры, так и сопроцессоры (то есть вычислительные устройства, которые не могут работать самостоятельно – только под управлением центрального процессора).

Определение 2. Гибридная вычислительная платформа – вычислительная система с распределенной памятью, каждый вычислительный узел которой является системой с общей памятью (обычно SMP).

Предлагаемый проект является развитием существующих учебных программ и курсов основной образовательной программы "Прикладные физика и математика, уровень магистратура.

Соответствует образовательным стандартам Санкт-Петербургского государственного университета и направлен на формирование следующих профессиональных компетенций:

1. Знание и понимание особенностей архитектур современных вычислительных систем и тенденций их развития.
2. Знание и понимание того, как особенности архитектур гибридных вычислительных систем влияют на эффективность программного обеспечения.
3. Знание технологий программирования высокопроизводительных вычислений для серверных процессоров, а также разных типов ускорителей вычислений.
4. Знание возможных причин некорректной работы программного обеспечения, причин потери эффективности и методов преодоления этих проблем.
5. Знание теоретических законов масштабируемости программного обеспечения.
6. Умение совместно использовать различные технологии разработки прикладного программного обеспечения для высокопроизводительных вычислений.
7. Умение оптимизировать программное обеспечение.

Целевая аудитория курса – студенты магистратуры Санкт-Петербургского государственного университета. Эта категория обучающихся имеет достаточную базу знаний по информационным технологиям, программированию, математике и предметной области своей научной специализации.

Предлагаемый к разработке курс состоит из следующих разделов:

1. Современные архитектуры вычислительных систем вообще и высокопроизводительных систем в частности, тенденции их развития. Современный серверный процессор. Графические процессоры общего назначения. Ускорители вычислений Intel® Xeon Phi. Другие типы сопроцессоров. Анализ рейтинга Top500 Supercomputers. Гибридные архитектуры первого и второго типа.
2. Принципы параллельного программирования, теоретический анализ эффективности и масштабируемости высокопроизводительных вычислений, теоретические модели параллельных вычислений и законы. Модель вычислений и закон Амдала. Законы Густафсона-Барсиса и Сана-Ная. Особенности программирования для мобильных и встраиваемых систем.
3. Факторы, влияющие на эффективность приложений. Мощность арифметических и логических выражений. Кэш-эффективность. Расположение данных. Предсказание ветвлений и другие аспекты. Необходимость контроля корректности программного обеспечения при его оптимизации.
4. Многопоточный параллелизм и его реализация на серверных процессорах. Особенности, проблемы эффективной реализации, условия их появления и способы разрешения. Краткий обзор технологий программирования, в том числе OpenMP. Обзор алгоритмов, которые эффективно отображаются на данную архитектуру.
5. Многопоточный параллелизм и его реализация на ускорителях вычислений. Учет особенностей архитектуры ускорителей, специфические проблемы эффективной реализации, условия их появления и способы разрешения. Концепция выгрузки вычислений. Обзор алгоритмов, которые эффективно отображаются на данную архитектуру.
6. Многопоточный параллелизм на гибридных архитектурах. Совместное использование разных технологий программирования. Декомпозиция алгоритма на часть, предназначенную для реализации на хост-процессоре и часть, реализуемую для сопроцессора.
7. Многозадачный параллелизм. Краткий обзор технологии программирования MPI. Многозадачность и многопоточность в одном приложении. Обзор алгоритмов, которые эффективно отображаются на архитектуру кластерных вычислительных систем. Декомпозиция алгоритма на часть, которая эффективно реализуется в рамках многозадачного параллелизма, и части многопоточного параллелизма разного вида. MPI вместе с OpenMP, потокобезопасность.

Методология преподавания данного курса предполагает тесную интеграцию нескольких форм методической работы:

1. Теоретические лекции.
2. Демонстрация теоретических положений с помощью заранее подготовленных относительно простых примеров программ.
3. Выполнение более сложных исследовательских заданий, содержанием которых является применение изучаемых подходов, исследование их эффективности, масштабируемости и иных свойств.

Особенности курса:

- Тесная интеграция вышеперечисленных трех компонентов (часто курсы по информационным технологиям включают только лекции или лекции и семинары, посвященные решению задач стандартного типа).
- Методическим сопровождением третьего компонента данного курса будут лабораторные работы, включающие описания, краткие инструкции преподавателю, шаблоны программ.
- Наполнение курса - предполагается рассматривать вопросы разработки эффективного программного обеспечения именно для гибридных архитектур.

Санкт-Петербургский государственный университет
Физический факультет
кафедра вычислительной физики

Отчёт по лабораторной работе

Реализация параллельных алгоритмов основных векторных операций с использованием OpenMP и программных инструментов Intel

выполнил Мартынюк Сергей Анатольевич
студент 423 группы

Описание работы

В работе предлагается исследовать эффективность выполнения алгоритмов векторных операций на примере вычисления скалярного произведения двух векторов очень больших размеров. Программы исполнялись на компьютере с Windows Server 2012, входящем в состав вычислительного кластера и имеющем два 8-ядерных процессора с технологией Hyper Threading, образующих в общей сложности 32 логических ядра, и 64 Гб оперативной памяти. Для исследования программ применялся Microsoft Visual Studio C 2012 с компилятором Intel Composer XE 13.0 в различных режимах. Целью настоящей работы является исследование скорости выполнения программы и её масштабируемости с применением различных технологий.

Во всех случаях, если не оговорено иное, программа компилировалась со вторым уровнем оптимизации /O2, и вычислялось скалярное произведение векторов, состоящих из 150 млн. вещественных элементов двойной точности. Для увеличения времени выполнения вычисление производилось 150 раз. Также были включены опции поддержки больших адресов и OpenMP для возможности запуска программы. Измерения времени проводились по 6–10 раз, и из полученных значений бралось несколько наименьших для последующей обработки. Такая методика была выбрана из-за переменной тактовой частоты процессоров, меняющейся от запуска к запуску программы.

Задание 1. Последовательная программа вычисления скалярного произведения

Последовательная программа компилировалась с опциями по умолчанию. Полученное время работы программы: (28,9±1,3) с. Вычисленное скалярное произведение: 291913,05.

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

#define N 150000000

double * a = new double[N];
double * b = new double[N];

int i, k;

double start, stop;
double gDotProduct;

int main()
{
using namespace std;
// initialize vectors

for (i = 0; i < N; i++)
{
    a[i] = 0.0941;
    b[i] = 0.08707;
}

cout << "Computed value of vector sum: " << endl;

start = omp_get_wtime();
for ( k = 0; k < 150; ++k )
{
    gDotProduct = 0.;
    for ( i = 0; i < N; ++i )
    {
        gDotProduct += a[i] * b[i];
    }
}

stop = omp_get_wtime();


//print dot product

cout << "sum = " << gDotProduct << endl;

//print delta time

cout << "time = " << stop - start << endl;
delete [] a;
delete [] b;
cin >> i;

}
```

Листинг 1. Последовательная программа вычисления скалярного произведения.

Задание 2. Многопоточная реализация вычисления скалярного произведения с OpenMP

Для распараллеливания программы перед циклом

```
for ( i = 0; i < N; ++i )
```

добавлена следующая строка

```
#pragma omp parallel for
```

Для использования возможности распараллеливания в настройках компилятора включена поддержка OpenMP. Результаты работы программы приведены в таблице 1.

число потоков	время, с	результат
1	(29,2±1,2)	291913
2	(12,5±0,4)	145957

Таблица 1

Полученные результаты показывают, что двухпоточная программа работает примерно в 2 раза быстрее по сравнению с однопоточной, но даёт неверный результат. Это связано с тем, что при распараллеливании цикла по умолчанию в результирующую переменную записывается значение локальной переменной первого потока, в которой находится только частичный результат. Значения локальных переменных других потоков с частичными результатами теряются. Для исправления программы требуется дополнительное описание параметров распараллеливания.

Задание 3. Многопоточная улучшенная реализация вычисления скалярного произведения с OpenMP

Исправление параллельной программы произведено путём замены строки

```
#pragma omp parallel for
```

на строку

```
#pragma omp parallel for reduction(+:gDotProduct)
```

Эта строка означает распараллеливание цикла и суммирование на заключительном этапе локальных переменных-копий, результатов работы потоков, в итоговый результат. После такого исправления программа даёт верный результат при произвольном числе потоков.

Число параллельных потоков задаётся подпрограммой:

```
omp_set_num_threads(num_threads);
```

В этом случае становится возможным исследование скорости работы программы в зависимости от числа потоков. Полученная зависимость представлена на рисунке 1.

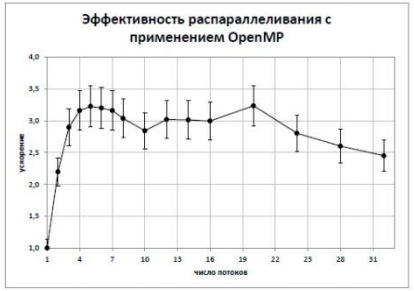


Рисунок 1. Ускорение работы программы от применения технологии OpenMP

Из графика на рисунке 1 видно, что ускорение растёт пропорционально числу потоков при малом количестве потоков. При числе потоков, большем 4, коэффициент ускорения перестаёт расти, достигая максимального значения 3,2. Когда количество потоков превышает 16, ускорение неуклонно падает с ростом многопоточности. Это связано с ростом дополнительных затрат при создании и уничтожении большого числа потоков. Также с увеличением числа потоков растёт вероятность замедления отдельного потока, попавшего на более загруженное ядро, завершения которого будут ожидать остальные потоки, закончившие свою работу быстрее. При числе потоков, превышающем 16, срабатывает влияние технологии HyperThreading, превращающей одно физическое ядро в два логических, но в данном случае она не повышает производительность программы.

Задание 4. Реализация программы вычисления скалярного произведения с использованием библиотеки Intel Math Kernel Library

Скалярное произведение вычислялось с помощью подпрограммы cblas_ddot:

```
gDotProduct = cblas_ddot(N, &a[0], 1, &b[0], 1);
```

Число потоков для выполнения подпрограмм Intel MKL задаётся следующим образом:

```
MKL_Set_Num_Threads(num_threads);
```

Наличие в библиотеке MKL возможности ручного задания числа потоков позволило провести исследование масштабируемости программы и сравнить эффективность библиотеки MKL с эффективностью технологии OpenMP. Результаты для библиотеки MKL показаны на рисунке 2.

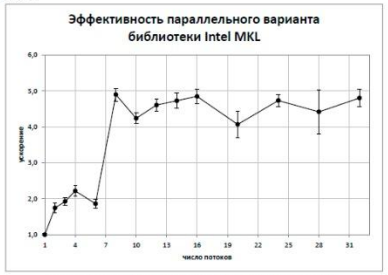


Рисунок 2. Ускорение работы программы от использования библиотеки MKL

Вначале скорость программы растёт даже медленнее, чем в случае с OpenMP, но когда число потоков достигает 8, коэффициент ускорения выходит на максимальное значение 4,9, заметно превосходя наибольший коэффициент для OpenMP, равный 3,2. Затем с дальнейшим увеличением количества потоков ускорение остаётся в пределах 4,1–4,8, не меняясь значительно. Лучший по сравнению с OpenMP для библиотеки Intel MKL результат объясняется её высокой оптимизацией с использованием распараллеливания и технологий аппаратного ускорения вычислений.

Задание 5. Использование расширений SSE и автоматического распараллеливания при компиляции для программы вычисления скалярного произведения

Эффективность оптимизаций компилятора исследовалась для последовательной программы, исходный код которой приведён в листинге 1. Данные о времени работы программы, созданной с применением различных опций компилятора, представлены в таблице 2.

оптимизация	время, с	прирост, %
O0	(69,7±1,0)	(0,0±0,0)
O1	(78,2±0,9)	(-10,9±0,2)
O2	(62,2±0,4)	(12,0±0,2)
SSE3, O2	(57,9±1,2)	(20,3±0,5)
AVX, O2	(58,8±1,0)	(18,5±0,4)
parallel, O2	(31,7±3,0)	(119,8±11,8)

Таблица 2

Оптимизация второго уровня O2 даёт прирост небольшой скорости на 12%. Использование векторных инструкций SSE3 и AVX при компиляции приводит к примерно одинаковому росту производительности на 20%. Технология автоматического распараллеливания показала наилучший результат, увеличив скорость работы программы в 2,2 раза.

Задание 6. Оптимизация программы вычисления скалярного произведения компилятором на основе профилирования

В данном случае используется профилировщик для анализа последовательной программы, поиска в ней «горячих точек» и последующей оптимизации с перекомпилацией. Время работы последовательной профилируемой программы (84,9±1,0) с. После выполнения профилирования и осуществления оптимизации время уменьшилось до (38,4±0,6) с, что соответствует увеличению скорости в 1,8 раза по сравнению с последовательной программой. Скорость работы оптимизированной профилировщиком программы близка к скорости программы, автоматически распараллеленной компилятором.

Задание 7. Реализация программы вычисления скалярного произведения с помощью Intel Integrated Performance Primitives

Вычисление выполнялось с помощью подпрограммы ippsDotProd_64f:

```
ippsDotProd_64f(a, b, (N * 1), &gDotProduct);
```

В программе использовался последовательный вариант библиотеки. Время работы (24,4±1,5) с. Программа с IPP работает на 18% быстрее последовательной версии. Величина прироста близка к значению, полученного при применении оптимизаций с использованием векторных инструкций SSE и AVX. Последовательная версия библиотеки Intel IPP оптимизирована под векторную архитектуру процессоров.

Вывод

Наибольшая эффективность программы достигается тогда, когда используются распараллеливание или высокопроизводительные библиотеки с параллельными подпрограммами и умеренная компиляторная оптимизация. Число запущенных потоков влияет на эффективность программы, поэтому оптимальное их количество соответствует числу процессорных ядер в системе.

Для реализации данного проекта необходим компьютерный класс, имеющий гибридную архитектуру и позволяющий продемонстрировать различные технологии программирования для таких систем.

Требуется также программное обеспечение, предназначенное для разработки – интегрированные среды, компиляторы, средства динамического анализа приложений. В Ресурсном образовательном центре «Физика» Санкт-Петербургского государственного университета имеется компьютерный класс, отвечающий этим требованиям.

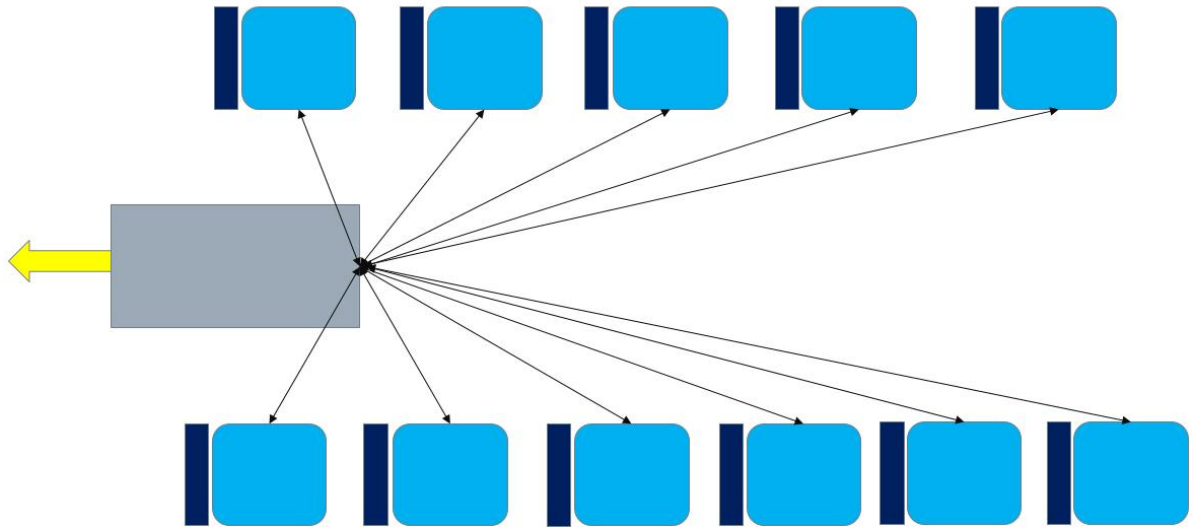
Компьютерный класс РОЦ «Физика» («Интел»)

11 мест => вычислительный кластер (многопроцессорная вычислительная система с гибридной архитектурой I и II типа).

1 рабочее место = 1 узел кластера = 1 или 2 процессора Intel® Xeon e5-2690, 2.9 ГГц (1 процессор = 8 физических ядер = 16 логических ядер), 16-64 гб оперативной памяти.

Быстрый интерконнект (коммутатор) – Gigabit Ethernet.

1 или 2 ускорителя вычислений на узел для выполнения гибридных вычислений: графический процессор общего назначения (технологии программирования CUDA, OpenCL и другие, offload – «выгрузка» вычислений) – Tesla C2075 и Intel® Xeon™ Phi (модель 3120).



5 * 16 + 6 * 32 = 272 логических ядра
+
Примерно 400 логических ядер Intel® Xeon™ Phi.
+
Примерно 18 * 480 = 8640 ядер GPGPU
=
Более 9000 вычислительных ядер.

Программная конфигурация

Виртуализация. Host – система: Microsoft Windows Server 2016.
+
Средства профессиональной разработки по от Microsoft, Intel и NVIDIA.
+
Средства анализа и оптимизации ПО. Средства программирования параллельных и высокопроизводительных вычислений. HPC Pack.

Примеры тем.

Изменение области выгрузки на основе потоков данных

Если выгружать все фрагменты, обладающие параллелизмом, может оказаться большое количество маленьких областей выгрузки. Их выбор и их количество должны быть сбалансированы с передачей данных с CPU на ускоритель и обратно. Скорость обмена данными зависит от скорости интерфейса PCI-E и может быть низкой. Транзакции могут быть также затруднены маршалингом (упаковкой) в прагме offload или другими факторами.

Если две параллельные секции разделены последовательной секцией, можно выбрать одну из двух стратегий:

a) Переместить выходные данные обратно на хост-процессор, выполнить последовательный код, переместить входные данные на устройство.
b) Оставить данные на устройстве и выполнить последовательный код, применив выгрузку к обеим параллельным секциям и последовательной секции, расположенной между ними.

Выбор механизма передачи данных

Модель Copyin/Copyout (#pragma offload)

Эта модель поддерживается компиляторами Intel C/C++ и Intel Fortran.
Если данные, передаваемые между CPU и сопроцессором – скаляры или массивы поразрядно копируемых элементов, следует выбирать модель #pragma offload. Эта модель требует локализованных изменений кода в точке выгрузки и изменения описаний функций. Программы на Fortran поддерживают только эту модель.

Модель общей памяти (_Cilk_shared/_Cilk_offload)

Эта модель поддерживается только компилятором Intel C/C++.
Если данные более сложные, чем скаляры или поразрядно копируемые массивы, следует использовать конструкции _Cilk_Shared/_Cilk_Offload. В этой модели функции и статически размещаемые данные должны описываться с атрибутом _Cilk_shared, а динамически размещаемые данные должны находиться в общей памяти. Реализация модели и использование _Cilk_Shared/_Cilk_Offload для модели программирования с общей памятью может быть более сложным, однако, класс программ, которые могут использовать архитектуру Intel MIC гораздо богаче, так как к этому классу принадлежат почти все программы C/C++.

Выгрузка с #pragma offload. Измерение производительности выгрузки

Накладные расходы на инициализацию

По умолчанию, когда программа выполняет первую директиву #pragma offload, инциализируются все устройства MIC, назначенные программе. Инициализация заключается в загрузке MIC программы на каждое устройство, создании канала передачи данных между CPU и устройством, создании MIC потока для обработки запросов выгрузки от потока CPU. Всё это требует времени. Следует исключить данные одноразовые накладные расходы, выполняя «dummy offload» на устройство.

```
// Пример пустой выгрузки для инициализации
int main()
{
#pragma offload_transfer target(mic) ... }
int main()
{
#pragma offload_transfer target(mic)
…
}
```

Альтернативно, можно использовать переменную окружения `OFFLOAD_INIT=on_start` для предварительной инициализации всех доступных MIC устройств.

Передача данных при выгрузке. Минимизация входных данных

Локальные вычисления, где это возможно.

Использование данных постоянного хранения при выгрузке

Если данные, сформированные в конце выгрузки требуются для следующей выгрузки, следует их оставить на сопроцессоре. Следующие выгрузки должны выполняться на тот же сопроцессор – использовать явное указание номера сопроцессора в операторе target.

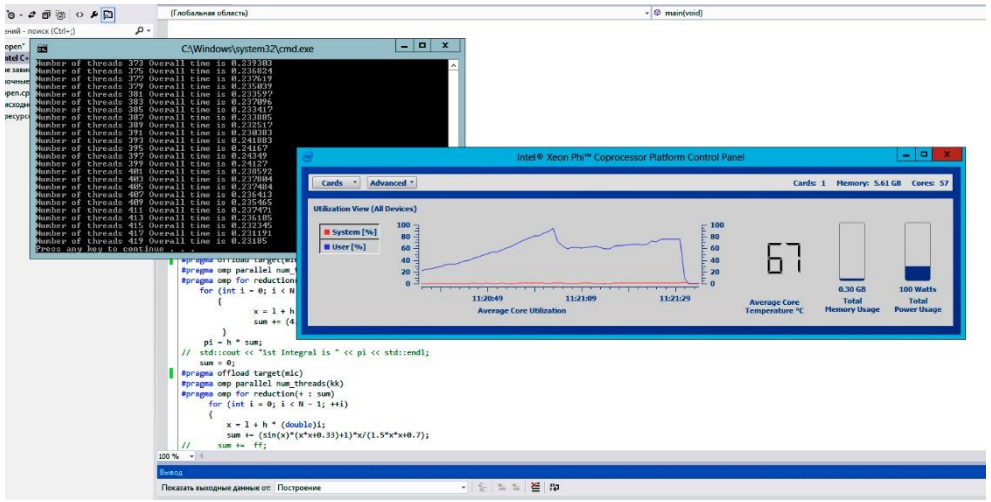
Длительное хранение: статически размещаемые данные

При использовании модификатора `nocopy()` автоматического выделения, а значит и удаления памяти не происходит. Это позволяет выделять память только в рамках сопроцессора и использовать ее для хранения данных между вызовами кода для Intel Xeon Phi.

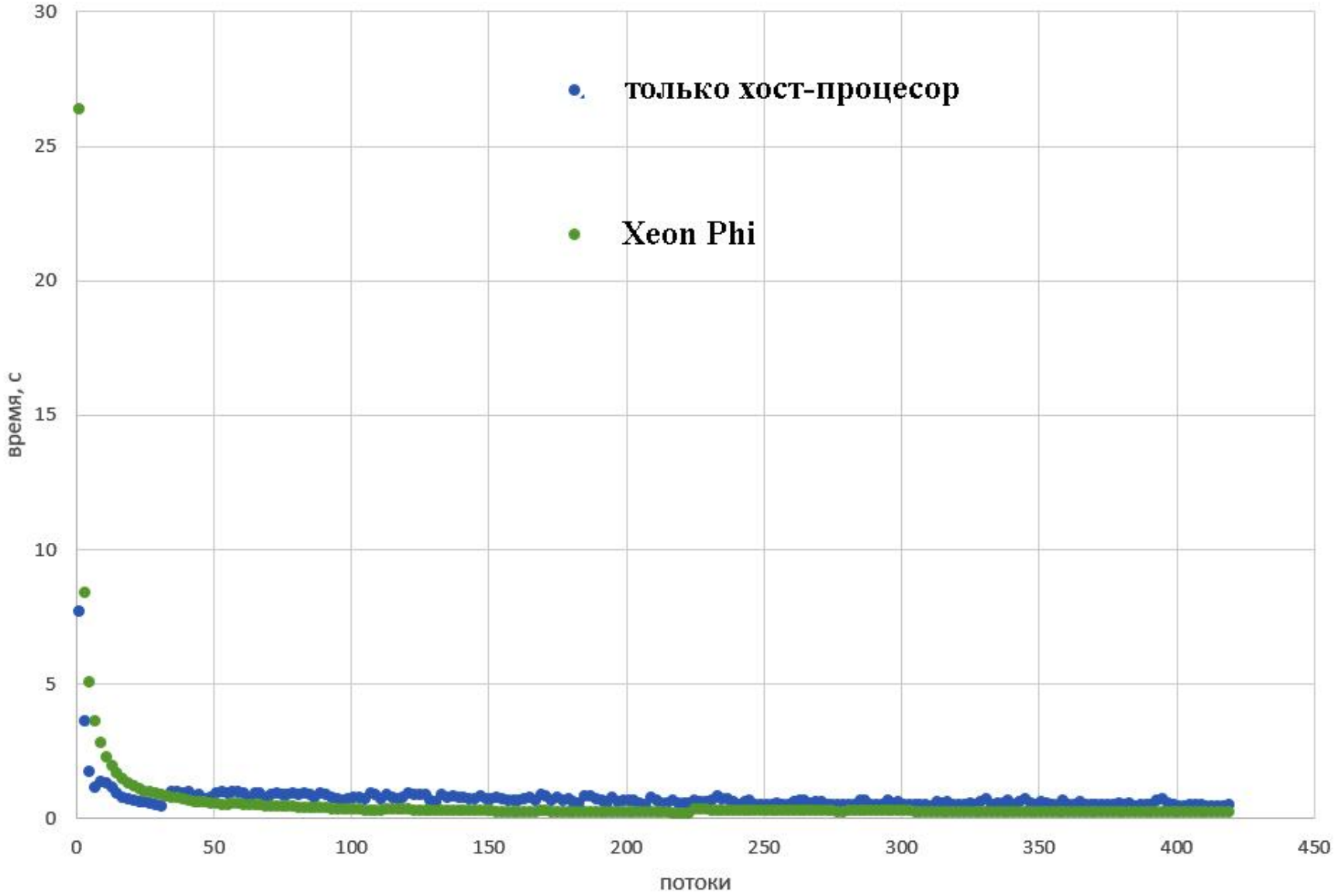
Пример. Вычисление интегралов. Явная выгрузка вычислений.

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

void main(void)
{
        double tim;
        double l, r, Integral=0;
        double x, ff, sum=0, pi, h;
        double nnn;
        long i, n=14, N=5000000000;
        l=0;
        r=1;
        h = (r - l) / (N - 1);
        for(int kk=1; kk<420; kk=kk+2)
        {
        tim=omp_get_wtime();                    sum = 0;
#pragma offload target(mic)
#pragma omp parallel num_threads(kk)
#pragma omp for reduction(+ : sum)
    for (int i = 0; i < N - 1;  ++i)
      {
            x = l + h * ((double)i - 0.5);
                  sum += (4.0 / (1.0 + x*x));
       }
    pi = h * sum;
```

```
        sum = 0;
#pragma offload target(mic)
#pragma omp parallel num_threads(kk)
#pragma omp for reduction(+ : sum)
        for (int i = 0; i < N - 1; ++i)
                {
                        x = l + h * (double)i;
                        sum += (sin(x)*(x*x+0.33)+1)*x/(1.5*x*x+0.7);
                }

        tim=omp_get_wtime()-tim;
        std::cout << "Number of threads " << kk << " Overall time is " <<
tim << std::endl;
        }
}
```

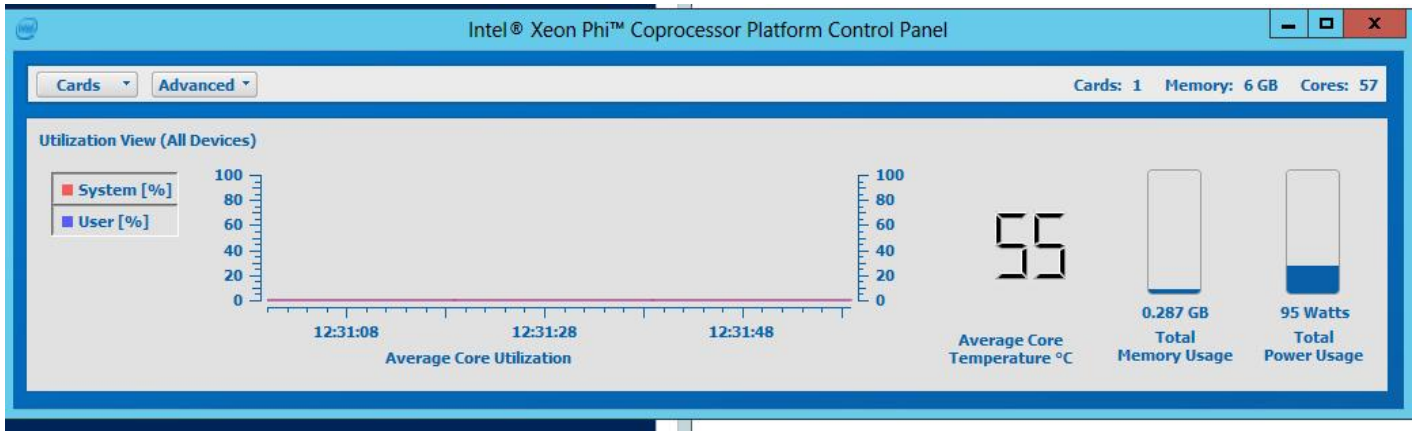


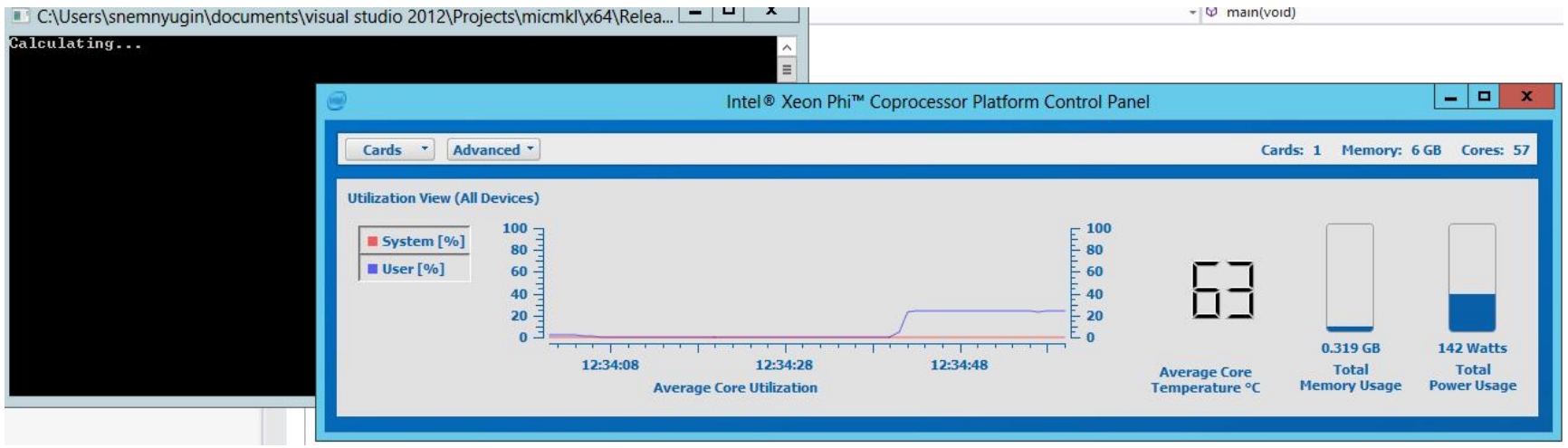
Тестовый пример. Вычисление интегралов.



Работа с Intel® Xeon™ Phi как повод поговорить о «зеленых» вычислениях. Экологизация вычислений.

Примеры увеличения энергопотребления при запуске приложения на сопроцессоре Intel® Xeon™ Phi.





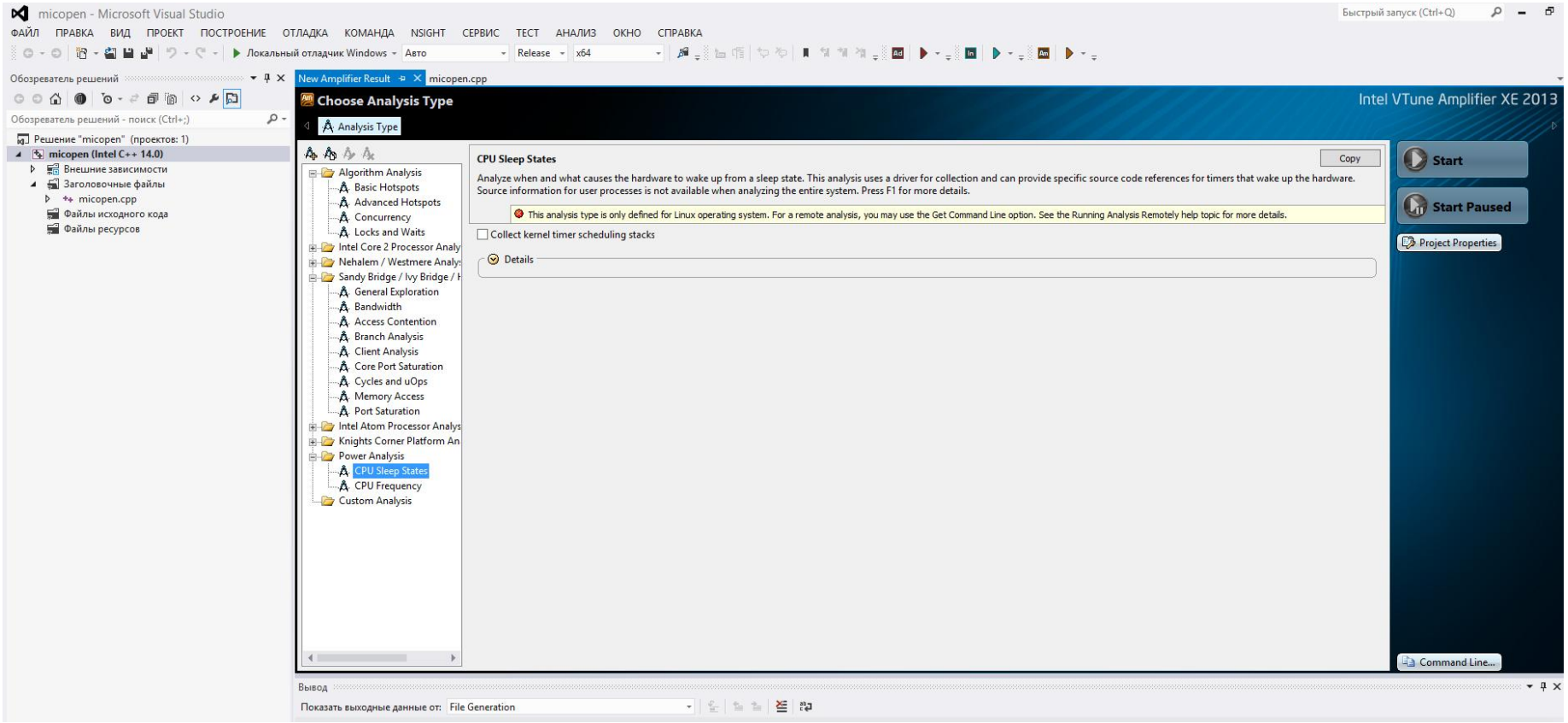
Как увеличить энергоэффективность программного обеспечения:

- ✓ Оптимизация передачи данных.
- ✓ Уменьшение вычислительной сложности алгоритмов.
- ✓ Использование «спящего» режима для периферийных устройств.
- ✓ Сбалансированная загрузка вычислительных узлов/ядер.
- ✓ Использование JIT-компиляции. Для интерпретируемых языков (основанных на генерации байт-кода) JIT-компиляция может увеличить эффективность программы.
- ✓ Применение компиляторной оптимизации
- ✓ Минимизация утечек памяти.
- ✓ Использование высокопроизводительных библиотек.
- ✓ Использование энергоэффективных алгоритмов.
- ✓ Оптимальное использование P- и C-состояний процессора.
- ✓ Применение многопоточности.
- ✓Использование Lazy loading (ленивая загрузка).
- ✓Эффективный UI (в том числе на базе исследования HCI – Human Computer Interactions).
- ✓Предпочтительное использование статического GUI (без анимации).
- ✓Оптимизация выполнения запросов (к БД, в Web-приложениях).

- ✓ Минимизация использования поллинга (регулярного опроса состояния устройства) или минимизация его частоту.
- ✓ Следует избегать использования байт-кода.
- ✓ Виртуализация позволяет строить большие энергоэффективные дата-центры, но с точки зрения выполнения прикладной программы она может быть неэффективной.
- ✓ Использовать меньшие значения тактовой частоты.
- ✓ Использовать ассемблер для часто используемых фрагментов кода. Низкоуровневое программирование.
- ✓ Использование специализированного аппаратного обеспечения.
- ✓ Тогда, когда это возможно, следует переводить части приложения в состояние сна.
- ✓ Пакетный ввод-вывод.
- ✓ Оптимизация использования периферийных устройств.
- ✓ Уменьшение избыточности данных (используется для повышения надежности хранения данных, но требует дополнительных транзакций и затрат энергии).
- ✓ Следует обращать внимание на правильную и своевременную обработку сигналов операционной системы. Наличие необработанных сигналов может препятствовать переходу системы в состояние с меньшим энергопотреблением.
- ✓ Уменьшение транспарентности (уровня абстракции).
- ✓ Динамическое уменьшение QoS (Quality of Service).
- ✓ Использование актуального ПО 3 стороны.
- ✓ Освобождение неиспользуемой памяти.
- ✓ Использование асинхронного ввода-вывода.
- ✓ Оптимизация взаимодействия между ПО и аппаратным обеспечением.

Инструментальные средства анализа энергопотребления.

Intel® VTune Amplifier – наиболее функциональный инструмент.



Благодарности

Ресурсный образовательный центр Санкт-Петербургского государственного университета по направлению «Физика».

Спасибо за внимание!