Отчёт

по

Контрольной Работе №1

Преподаватель: Кулябов Дмитрий Сергеевич

Выполнил: Кремер Илья

Группа: НК-401

Москва, РУДН – 2011

Оглавление

Введение

Этот отчёт содержит только самые необходимые комментарии и соответствует последовательности выполнения работы.

Теоретическое задание 1: Классификация Флинна

Классификация Флинна – это общая классификация архитектур вычислительных машин по признакам наличия параллелизма в потоках команд и данных, разработанная профессором Стэндфордского Университета Майклом Флинном.

Проще всего представить таксономию в виде таблицы:

	Одиночный поток команд (**S**ingle **I**nstruction)	Множество потоков команд (**M**ultiple **I**nstruction)
Одиночный поток данных (**S**ingle **D**ata)	**SISD** (ОКОД)	**MISD** (МКОД)
Множество потоков Данных (**M**ultiple **D**ata)	**SIMD** (ОКМД)	**MIMD** (МКМД)

К SIMD относятся векторные архитектуры (выполнение одной операции сразу над многими данными – элементами вектора), к MISD – конвейерные ЭВМ, хотя существует мнение, что в чистом виде машин такого класса нет. К классу MIMD относят многопроцессорные системы, в которых процессоры обрабатывают множественные потоки данных. К классу SISD – классические последовательные машины, в которых есть только один поток данных, команды выполняются друг за другом и каждая команда инициирует только одну операцию с каждым потоком данных.

Классификация Флинна не является очень точной и строгой. Например, конвейерные ЭВМ могут быть отнесены и к классу SISD, и к классу SIMD (векторный поток данных с конвейерным процессором), и классу MISD (множество процессоров конвейера обрабатывают один поток данных последовательно), и классу MIMD – выполнение последовательности различных команд над множественным скалярным потоком данных.

Теоретическое задание 2: Архитектура кластера

В настоящее время кластер (суперкомпьютер) состоит из вычислительных узлов на базе стандартных процессоров, т.е. тех, что используются и в ПК. Узлы объединены сетью (интерконнектом), а также, как правило, вспомогательной и сервисной сетями. Большинство кластерных систем списка используют процессоры Intel (Intel Xeon, Intel Xeon EM64T, Intel Itanium 2), кроме них часто встречаются процессоры IMB: Power и PowerPC. В последнее время популярностью пользуются процессоры AMD (особенно AMD Opteron и его недавно вышедшая двухъядерная версия).

Узлы – это чаще всего двухпроцессорные SMP-серверы, собранные в 19-дюймовые стойки. Каждый узел работает под управлением своей копии ОС, чаще всего используется Linux. Состав и мощность узлов могут быть разными в рамках одного кластера, однако чаще кластер состоит из одинаковых по мощности узлов. Для интерконнекта применяются технологии Gigabit Ethernet, SCI, Myrinet, QsNet, InfiniBand и др.

Производительность кластера зависит не только от характеристик его процессоров и памяти, но и от архитектуры, системной шины и интерконнекта. Более того, для разных задач различные из этих факторов играют большую или меньшую роль. Так, например для рендеринга независимых сюжетов в видео (и других хорошо распараллеливаемых задач) основной фактор – это мощность процессоров, а интерконнект уходит на второй план. В то же время для множества задач аэродинамики увеличение числа узлов в кластере может мало повлиять на скорость решения задачи при слабой производительности системной сети.

Важным фактором, который является главным тормозом системы является скорость чтения данных извне, так что для эффективного доступа к данным чаще всего используется вспомогательная сеть (как правило Gigabit Ethernet с помощью каналов Fibre Channel).

Важную роль в архитектуре кластеров играют системы охлаждения и электропитания. Некоторые суперкомпьютеры потребляют более 90 кВт мощности, из которых почти все уходят в тепло, а весь кластер состоит из 288 узлов в корпусах 1U, заключённых в восьми стойках. В первых СК использовалось жидкостное охлаждение, которое со временем было вытеснено более продуманным дизайном стоек и охлаждением всего помещения, в котором расположен кластер.

Задача 1: Определённый интеграл с помощью формулы трапеции

По условию нам разрешается выбирать любую функцию для тестирования программы. Выберем сначала параболу и другие известные и простые функции, чтобы убедиться, что программа работает корректно (например, от `-1` до `1` интеграл от x^2 равен `2/3`, и т.д.).

На нас лежит выбор, каким образом регулировать точность вычисления интеграла. Возьмём количество интервалов (т.е. трапеций) и будем вычислять шаг как `(b - a)/(количество интервалов)`. Это место в коде послужит стартовой точкой для вычисления времени его работы.

```
s = 0
x1 = a; x2 = a + h
do i = 1, intervals
    s = s + h * ((func(x2) + func(x1))/2.0)
    x1 = x1 + h
    x2 = x2 + h
enddo
```

Так будет выглядеть подсчёт интеграла, теперь необходимо распараллелить этот цикл.

Здесь видно, что цикл вообще не зависит от `i` (итератор не встречается в теле цикла), однако на каждой итерации `x1` и `x2` принимают различные значения, поэтому придётся немного переделать цикл с тем, чтобы на любой итерации можно было вычислить свойственные ей `x1` и `x2`:

```
do i = 0, intervals - 1
    s = s + h * ((func(a + i * h) + func(a + (i + 1) * h))/2.0)
enddo
```

Точно такой же цикл можно использовать и в предыдущем варианте, однако в этом происходит больше операций (сложений и умножений).

Распараллелим этот цикл, используя директиву `reduction`:

```
!$omp parallel num_threads(8) firstprivate(a, b, h)
!$omp do reduction (+ : s)
do i = 0, intervals - 1
    s = s + h * ((func(a + i * h) + func(a + (i + 1) * h))/2.0)
enddo
!$omp end parallel
```

По алгоритму переменные `a`, `b` и `h` могут быть общими для всех процессов (т.е. объявлены как `shared`), однако это вызовет гонку данных –

множественный доступ к этим переменным из разных процессов. Объявление `private` собьёт имеющиеся значения, поэтому используем `firstprivate`.

Теперь возьмём функцию, которая удобна для большого количества интервалов, например, это синус, т.к. она является периодической с достаточно небольшим периодом. Возьмём $sin(x^2) + 5$, чтобы значение интеграла росло с увеличением расстояния между границами.

Будем тестировать программы, подставляя следующие значения:

a	b	intervals
–100.000	100.000	200.000.000

Сначала следует обратить внимание, что не распараллеленный вариант (файл `integral_wo_omp.f90`) даёт 2% прирост производительности по сравнению с распараллеленным, но запущенным с 1-им процессом: 83.0 секунды против 84.7[1].

Далее, как и ожидалось, распараллеленный алгоритм уходит сильно вперёд, т.к. способен использовать все предоставленные ему ресурсы на 100%:

Нитей	Время, сек	Загрузка CPU, %
2	42,4	25
4	17,0	50
8	9,75	100

Заменив `firstprivate` на `shared` получим 3% убыль в скорости (проверялось только для 4-ёх процессов).

Задача 2: Программа перемножения матриц

Используем цикл перемножения матриц:

```
do i = 1, N
    do j = 1, N
        c(i, j) = 0.0
        do k = 1, N
            c(i, j) = c(i, j) + a(i, k) * b(k, j)
```

[1] Было произведено несколько десятков запусков, поэтому разница не является случайной.

```
            end do
        end do
    end do
```

А для того, чтобы добиться распараллеливания просто поставим директиву `do` прямо над внешним циклом:

```
!$omp parallel do schedule(runtime) shared(a, b, c)
```

Итераторы автоматически объявляются как `private`, как и в прошлой задаче. В книге Антонова «Параллельное программирование с использованием технологии OpenMP» приведены временные результаты этой программы, запущенной на узле суперкомпьютера СКИФ МГУ «ЧЕБЫШЁВ». На обычном современном ПК скорость выполнения этой программы почти пропорциональна этим результатам.

Задача 3: Вычисление арксинуса с помощью ряда Тейлора

Все задачи на вычисление какой-либо функции через ряд Тейлора решаются следующим образом:

1. Находится отношение членов $n + 1$ и n в виде некой формулы.
2. В сумму, которая будет содержать конечный результат добавляется первый член (по формуле ряда).
3. Заводится переменная «предыдущий», в которую также кладётся значение первого члена ряда.
4. Начинается цикл от 0 до какого-то максимального разрешённого количества слагаемых (в ряде Тейлора сумма идёт от 0 до бесконечности), в котором сначала, как умножение предыдущего члена на отношение 1-ого к нулевому (цикл с нуля идёт из-за программной реализации), вычисляется текущий член и добавляется в сумму. Значение переменной «предыдущий» устанавливается на то, которое было добавлено в сумму.

Проиллюстрируем описанный алгоритм следующим общим примером:

Пусть $M = A[i+1]/A[i]$ (в общем виде), тогда сумма, например первых четырёх элементов будет равна:

$$S_4 = A1 + A1*M + A1*M*M + A1*M*M*M$$

`M` будет вычисляться на каждом шаге (зависит от номера шага).

Такой подход выглядит странно на первый взгляд, т.к. то же самое можно вычислить «в лоб» по имеющейся формуле, однако несложно понять, что для больших `i` (например, в несколько десятков) вычисление окажется крайне затруднительным из-за наличия в формуле факториалов, степеней и их произведений.

Перейдём к нашей формуле:

$$\arcsin x = x + \frac{x^3}{6} + \frac{3x^5}{40} + \cdots = \sum_{n=0}^{\infty} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1}$$

Чтобы вычислить 100-ый член, нам, по крайней мере, придётся считать $(100!)^2$, хотя очевидно, что истинное значение сотого члена очень мало.

Итак, вычислив отношение, напишем цикл:

```
do i = 0, N_MAX - 1
    m = ((2*i + 2) * (2*i + 1) * (2*i + 1) * X * X + 0.0) / &
         (4 * (i + 1) * (i + 1) * (2*i + 3) + 0.0)
    arcsin = arcsin + prev * m
    prev = prev * m
enddo
```

Следует избавиться от повторного вычисления `prev * m`. Для этого введём ещё одну переменную:

```
do i = 0, N_MAX - 1
    m = ((2*i + 2) * (2*i + 1) * (2*i + 1) * X * X + 0.0) / &
         (4 * (i + 1) * (i + 1) * (2*i + 3) + 0.0)
    tmp = prev * m
    arcsin = arcsin + tmp
    prev = tmp
enddo
```

Однако, запустив такую программу для `N_MAX = 2000` получим явно ошибочный результат, для `X = 1` это будет `1,92`, что превышает значение $\pi/2 \approx 1{,}57$. Эта ошибка – результат больших произведений в дроби. Если мы в строчку вычисления `m` на место `i` подставим `1999`, и выпишем её в коде отдельно, то программа не скомпилируется, выдав ошибку:

```
Error: Arithmetic overflow
```

Поэтому нам придётся вручную разбить эту дробь, чтобы избежать огромных чисел:

```
m = ((2*i + 2) + 0.0) / (4 * (i + 1))
```

```
m = (m * (2 * i + 1)) / (i + 1)
m = (m * (2*i + 1) * X * X) / (2*i + 3)
```

Теперь при любых N_MAX, необходимых для адекватной оценки времени работы алгоритма (хотя бы десятки тысяч) программа выдаст результат, ожидаемый в теории. Так, для 2.000 это будет примерно 1.5582, а для 100.000 – 1.5690 (для единицы ряд очень медленно сходится).

Остаётся доделать задание, сделав возможность вычислять арксинус с точностью до заданного ε.

Сделаем это следующим образом: сначала вычислим арксинус с помощью встроенной функции:

```
true_arcsin = asin(X)
```

В цикле будем каждый раз вычитать из этого значения текущий результат программного вычисления и при достижении меньшей, чем ε разности прекращать работу:

```
if (true_arcsin - arcsin <= EPS) exit
```

Если эта разница так и не будет достигнута, то программа просто израсходует все итерации, так и не вычислив арксинус с желанной точностью.

Протестируем программу на скорость со следующими параметрами:

N_MAX	EPS	X
10.000.000	0.00025	1

Для таких данных алгоритм проходит более половины итераций, выдавая конечный ответ: arcsin = 1,57079633, что ровно на ε меньше истинного значения. Это объясняется тем, что для таких больших номеров членов, сами значения становятся крайне малыми.

Теперь нам необходимо распараллелить алгоритм.

Очевидно, что в данном цикле каждая следующая итерация зависит от предыдущей, поэтому просто воспользоваться директивой do, как это было сделано в предыдущих задачах нам не получится.

Однако можно заметить, что формула отношения любого последующего члена к предыдущему не зависит ни от чего, кроме номера члена в последовательности. Следовательно, можно завести массив `m(N_MAX)`, заранее вычислить все отношения, и уже используя массив, вычислить саму сумму. Сразу же распараллелим цикл, вычисляющий все `m`:

```
!$omp parallel num_threads(8) shared(m)
!$omp do schedule(guided, 1000)
    do i = 0, N_MAX - 1
        m(i+1) = ((2*i + 2) + 0.0)/(4 * (i + 1))
        m(i+1) = (m(i+1) * (2 * i + 1)) / (i + 1)
        m(i+1) = (m(i+1) * (2*i + 1) * X * X) / (2*i + 3)
    enddo
!$omp end parallel
```

Следует отметить, что для замера было специально ограничено число `N_MAX` до `5.100.000`, т.к. иначе вычислений было бы заведомо больше, чем в не распараллеленном варианте, который бросает вычислять, достигая точности в то же цикле, где вычисляются и отношения.

Распараллеленный вариант оказывается всегда выигрышным, если считать не с точностью до заданного значения, а просто используя какое-то ограничение на количество слагаемых.

При использовании 8-ми процессов (максимум для данной машины) прирост производительности составил более 10%.

Особенного комментария заслуживает использование динамического выделения памяти для массива `m`. Оно используется из-за ошибки сегментации[2], возникающей, если объявлять `m` с большим количеством элементов (более 250 тысяч в Windows и более миллиона в Linux).

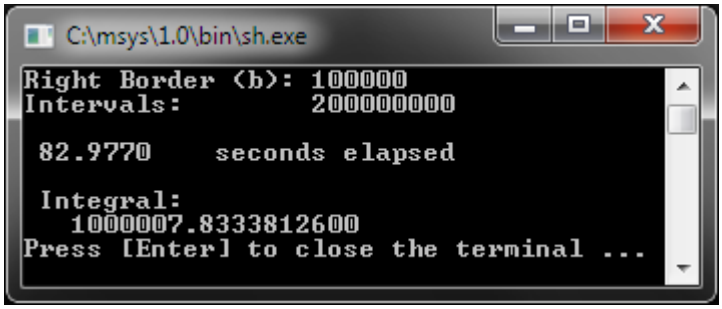
Ещё один интересный момент связан с вычислением `X * X` на каждой итерации. Хочется ввести переменную `sqrx` и сразу сохранить в ней значение `X * X`, однако это не то что бы не даст прироста производительности, а наоборот чуть замедлит алгоритм. Дело в том, что делая `X` константой (`paramter`), фрагмент всех выражений `(X * X)` всё равно вычисляется на этапе компиляции.

[2] В ОС Windows: Win32 OS Exception: c00000fd (stack overflow)

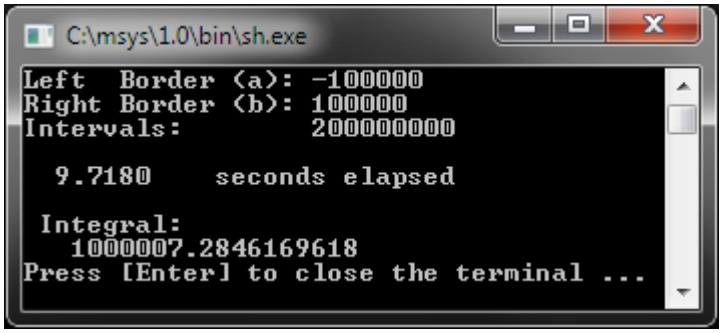
Иллюстрации выполнения программ[3]

Снимок экрана с выполнением программы перемножения матриц не включён в отчёт, т.к. матрицы инициируются случайным образом и отсутствует возможность вывести их куда-либо не в файл из-за огромных порядков.

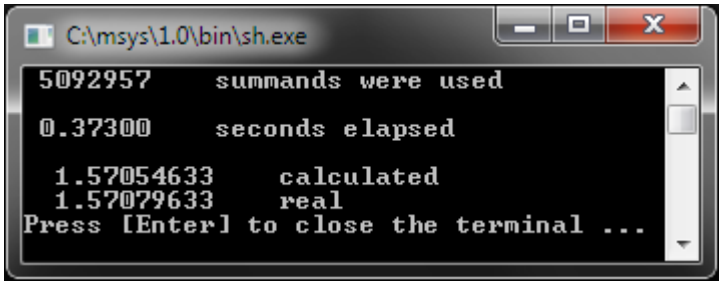
В исходных кодах указано, какие изменения необходимо сделать, чтобы протестировать корректность работы программы, выведя матрицы в терминал.



Линейное вычисление интеграла

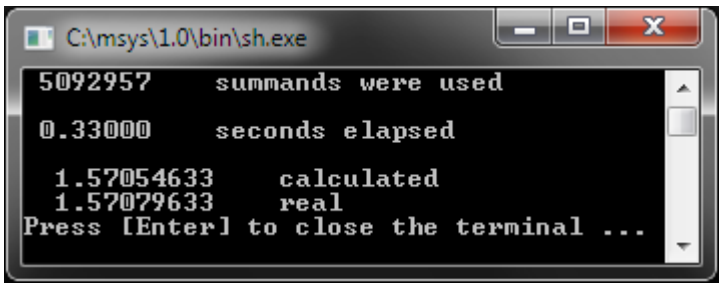


Параллельное вычисление интеграла



Линейное вычисление арксинуса

[3] Из-за проблем с кодировкой в командной строке Windows, в программах была использована латиница, а прикреплённые в контрольной работе файлы имеют другие тексты выдачи результатов.



Параллельное вычисление арксинуса

Неиспользование `sqrx` даст прирост производительности ещё на 2-3%.

Литература и др. источники

1. А.С. Антонов, «Параллельное программирование с использованием технологии OpenMP», – издательство Московского Университета, 2009.
2. http://parallel.ru – Информационно-аналитический центр.
3. http://ru.wikipedia.org/ – Информация о ряде Тейлора.
4. http://ru.wikipedia.org/ – Классификация Флинна
5. http://www.npk.ru/ – журнал «Upgrade» # 4 (23) 2005, Кластерные системы, часть 3 «Архитектура кластерных систем».