Отчёт

по

Контрольной Работе №2

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. Кроме того, существуют ссылки на предыдущий отчёт (Контрольная №1).

Программы не тестировались на скорость, т.к. этого не требуется в заданиях.

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)`.

```
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, используя MPI.

Используем следующую схему:

1. Каждый процесс забирает свой кусок из интервала `[a; b]`, т.е. вычитывает свою стартовую точку (`a`) и конечную (`b`).
2. Каждый процесс считает площадь подынтегральной функции на этом куске, а затем нулевой процесс суммирует все полученные результаты.

Чтобы упросить сбор результатов используем редукцию: `mpi_reduce`, это избавит нас от нужды посылать сообщения.

Остаётся только решить, как заставить каждый процесс работать со своим отрезком. Сделаем это следующим образом:

```
allocate(intervals_pp(np))
call distribute_intervals(intervals_pp, INTERVALS, np)
!intervals_pp = INTERVALS ! игнорирование распределения
!(увеличение точности с ростом np)

if (proc_id == 0) then
    print '(1x, a, 1x, i4)', "Интервалов задано:", INTERVALS
    print '(1x, a, 8x, f8.6)', "Средний шаг:", (b - a)/INTERVALS
    print *
    print *, "Кто что считает: (proc_id, a, b, n)"
endif
call mpi_barrier(MPI_COMM_WORLD, error)

part_len = (b - a)/np
svoe_a = a + proc_id * part_len
svoe_b = svoe_a + part_len
print '(i3, 4x, f8.6, 4x, f8.6, 4x, i4)', proc_id, svoe_a,
svoe_b, intervals_pp(proc_id + 1)
call mpi_barrier(MPI_COMM_WORLD, error)

s = 0
h = (svoe_b - svoe_a)/intervals_pp(proc_id + 1)
x1 = svoe_a; x2 = svoe_a + h
do i = 1, intervals_pp(proc_id + 1)
    s = s + h * ((sin(x2) + sin(x1))/2.0)
    x1 = x1 + h
    x2 = x2 + h
enddo

deallocate(intervals_pp)
```

Сначала мы выделяем небольшой массив под значения количества интервалов для каждого процесса (`np` – количество процессов). Далее, мы пишем функцию `distribute_intervals`, которая занимается распределением заданного количества интервалов. Например, задано 100 интервалов при 3-ёх процессах, результат её работы будет массив из чисел 34, 33 и 33.

Конечно можно просто завести одну переменную и написать что-то вроде: `intervals_pp = int(INTERVALS/np)`, но просто тогда мы неправильно берём исходное количество интервалов (например, было `100`, стало `33 + 33 + 33 = 99`), которое было бы задействовано в нераспараллеленном варианте.

Каждый процесс использует только одну переменную из этого массива, но так просто гораздо проще запрограммировать это распределение. Плюс, сам массив так или иначе очень мал и задача распределения крайне мала по затратам в сравнении с задачей подсчёта интеграла.

Далее мы предупреждаем, что нелогично задавать количество процессов большее, чем количество интервалов.

Используем барьеры для синхронизации печати – чтобы заголовок не напечатался в середине и для сохранения целостности блоков вывода (попробуйте убрать любой из барьеров).

Ищем длину отрезка (в каждом процессе свою) и локальные для процесса `a` и `b`, выводим эти данные.

Используем их для подсчёта куска интеграла, выводим локальную сумму, а затем встречаем на своём пути редьюс:

```
call mpi_reduce(s, total, 1, MPI_REAL, MPI_SUM, 0, MPI_COMM_WORLD, error)
```

Складываем все `s` в `total`, получая тем самым в нулевом процессе конечный ответ.

Задача 2: Вычисление следующего произведения с комплексным числом

$$(z+1)\prod_{k=1}^{n}\left(z^2+2z\cos\frac{2k\pi}{2n+1}+1\right), z \in \mathbb{C}$$

Необходимо проверить, что полученное число равно:

$$z^{2n+1}+1$$

Алгоритм задачи достаточно прост, запрограммируем эту формулу «в лоб»:

```
p = 1
do k = 1, N
    p = p * (2 * z * cos((2 * k * PI)/(2 * N + 1)) + 1 + z * z)
enddo

res = (z + 1) * p
```

Сравним `res` с тем, что предлагает вторая формула:

```
print *, res, "res"
print *, (z ** (2 * N + 1)) + 1
```

Видим, что для некоторых наборов данных, особенно для тех, которых требует проверка на скорость выполнения, происходит ошибка, связанная с ограничениями числа типа `double precision`. Это вполне ожидаемо, ведь возводить число `double`, например, в 1000-ую степень обычно не приходится.

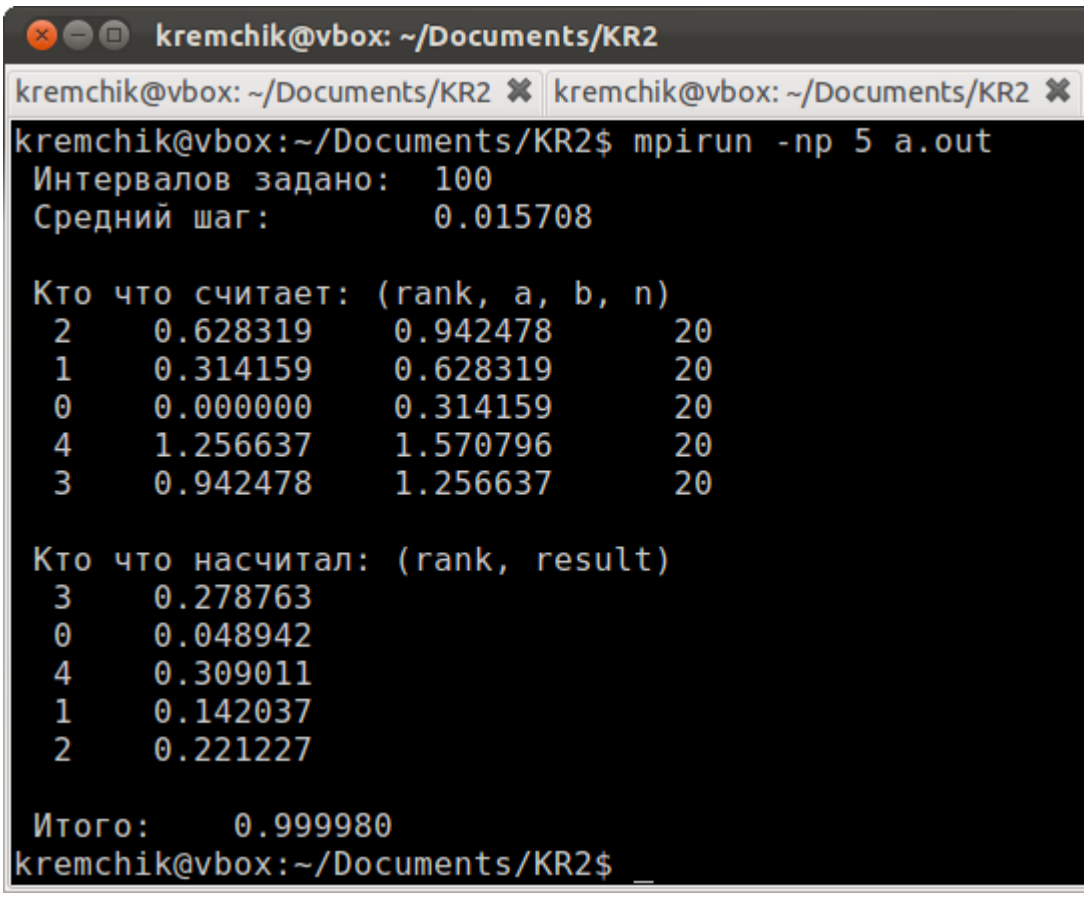
Распараллелим этот алгоритм, применив редукцию, как и в первой задаче.

```
p_pp = 1
do k = 1 + proc_id, N, nprocs
    p_pp = p_pp * (2 * z * cos((2 * k * PI)/(2 * N + 1)) + 1 + z
* z)
enddo
call mpi_reduce(p_pp, p_total, 1, MPI_COMPLEX, MPI_PROD, 0,
MPI_COMM_WORLD, error)
```

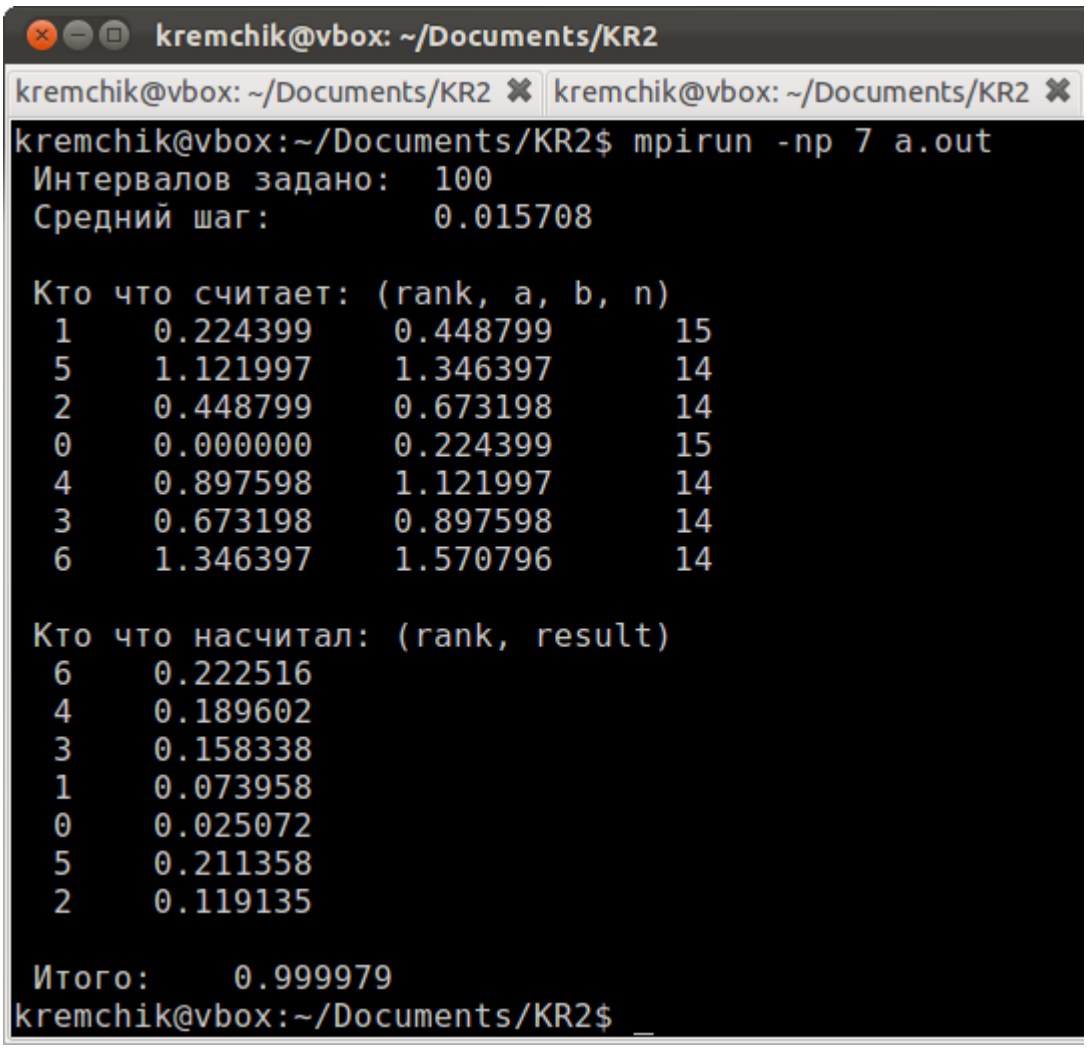
`p_pp` – это какая-то часть всего произведения, которая будет вычислена каждым процессом. Так что останется только встретить reduce, который соберёт эти части в одно произведением – `p_total`. Как обычно, работу сбора предоставляем нулевому процессу, т.к. он всегда существует (программа будет работать корректно и на одном процессе).

Иллюстрации выполнения программ

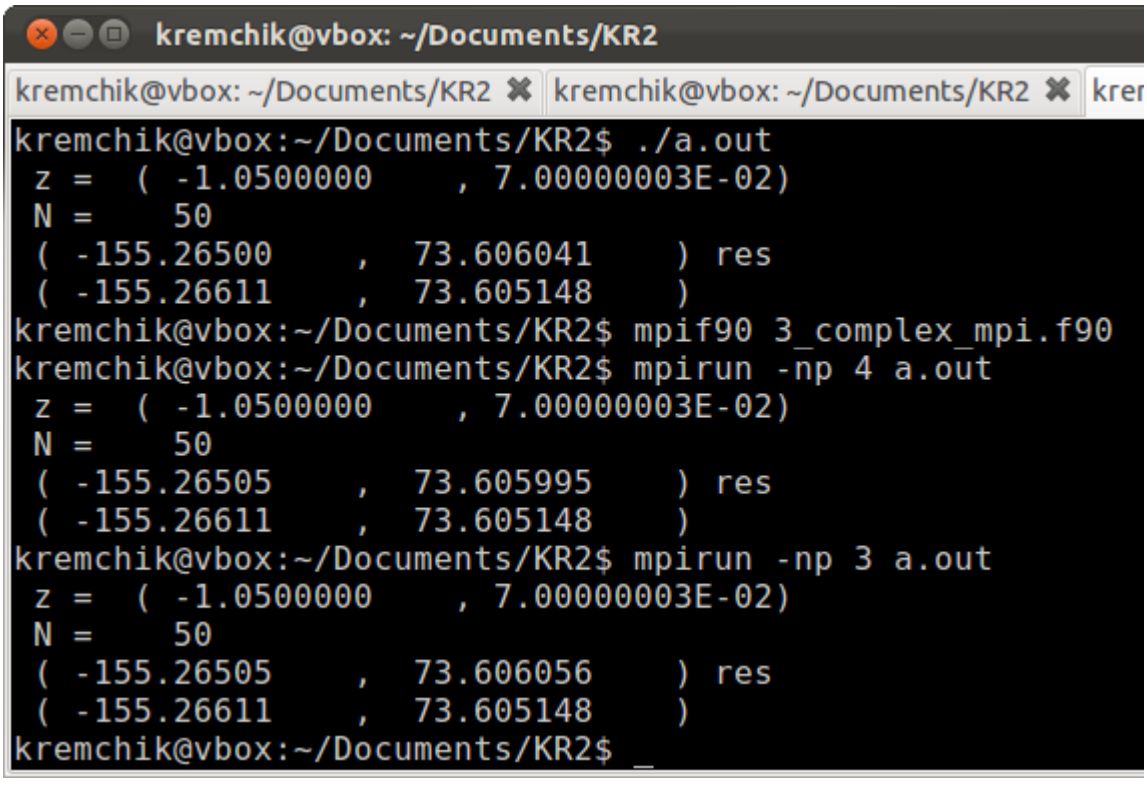
Снимки ко второй задаче и к первой части первой не сделаны, т.к. их содержимое очевидно (перемножение матриц и обычный подсчёт интеграла), а тесты на скорость не производились.



Вычисление интеграла, 5 процессов



Вычисление интеграла, 7 процессов (те же параметры)



Задача №3. Сначала нераспараллеленная версия, затем распараллеленная, запущенная с 4 и 3 процессами

Можно наблюдать различные результаты, как в интегральной, так и в последней задаче – это связано с погрешностями вычислений.

Литература и др. источники

1. А.С. Антонов, «Параллельное программирование с использованием технологии MPI», – издательство Московского Университета, 2009.
2. http://parallel.ru – Информационно-аналитический центр.