Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого
—
Институт компьютерных наук и технологий
Высшая школа искусственного интеллекта

ЛАБОРАТОРНАЯ РАБОТА

«Интегрирование методом Монте-Карло»

по дисциплине «Параллельное программирование на суперкомпьютерных системах»

Выполнил: студент группы
3540201/20302 Тугай А.А.

<подпись>

Проверил: Лукашин А.А.
доцент, к.т.н.

<подпись>

Санкт-Петербург
2023

ПОСТАНОВКА ЗАДАЧИ

1. Выбрать задачу и проработать реализацию метода Монте Карло для вычисления интеграла, допускающего распараллеливание на несколько потоков/процессов.
2. Разработать тесты для проверки корректности алгоритма (входные данные, выходные данные, код для сравнения результатов). Для подготовки наборов тестов можно использовать математические пакеты, например, matlab (есть в классе СКЦ и на самом СКЦ).
3. Реализовать алгоритмы с использованием выбранных технологий.
4. Провести исследование эффекта от использования многоядерности / многопоточности / многопроцессности на СКЦ, варьируя узлы от 1 до 4 (для MPI) и варьируя количество процессов / потоков.
5. Подготовить отчет в электронном виде.

Прикладная задача: Интегрирование методом монте-карло

Технологии:

1. C & Linux pthreads
2. C & MPI
3. Python & MPI
4. C & OpenMP

1 ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Метод Монте-Карло — это статистический численный метод, который используется для приближенного вычисления интегралов, основываясь на генерации случайных чисел.

1.1 Основной алгоритм метода Монте-Карло

Пусть требуется вычислить определённый интеграл:

$$\int_a^b f(x)\,dx$$

Рассмотрим случайную величину u, равномерно распределённую на отрезке интегрирования [a,b]. Тогда f(u) также будет случайной величиной, причём её математическое ожидание выражается как:

$$\mathbb{E}f(u) = \int_a^b f(x)\varphi(x)\,dx,$$

где $\varphi(x)$ — плотность распределения случайной величины u, равная $\frac{1}{b-a}$ на участке [a,b]. Таким образом, искомый интеграл выражается как:

$$\int_a^b f(x)\,dx = (b-a)\mathbb{E}f(u),$$

где матожидание f(u) вычисляется по формуле:

$$\frac{1}{N}\sum_{i=1}^{N} f(u_i)$$

Таким образом, получаем оценку интеграла:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{N}\sum_{i=1}^{N} f(u_i).$$

Точность оценки зависит только от количества точек N.

1.2 Возможность распределения вычислений

Метод Монте-Карло может быть легко распараллелен, что позволяет ускорить вычисления на многопроцессорных системах или в распределенных вычислительных сетях.

В распараллеленном методе Монте-Карло, вместо генерации случайных точек на одном процессоре, точки генерируются на нескольких процессорах или узлах вычислительной сети. Затем значения функции в каждой точке вычисляются на каждом процессоре. Далее, значения функции с каждого процессора собираются на главном процессоре и усредняются, чтобы получить приближенное значение интеграла.

При правильном распределении задач между процессорами возможно достичь ускорения вычислений на несколько порядков по сравнению с последовательной версией метода Монте-Карло.

Так как точки распределяются случайным образом, распределение задач между потоками не требует специального алгоритма, но требует инициализации ГПСЧ уникальным сидом.

2 РЕАЛИЗАЦИЯ

Для создания тестовых данных был использован Matlab и реализован скрипт приведенный в Приложении 1. Результатом работы является таблица где a, b - интегрирования, a result – соответственно вычисленное значение интеграла от функции $(sin(x)\,cos(x))^2 - \sqrt{6x}$.

Результат работы скрипта представлен Рисунке 1:

a	b	result
0.1	0.19	-0.081848
3.5	9.2	-33.319
17.8	95.4	-1380
54.2	97.5	-909.57
56.9	81.7	-498.75
9.3	38.5	-336.69
9.1	321.2	-9277.8
12.4	23.1	-107.58

Рисунок 1 – сгенерированные тесты.

Проведенный тест показал, что алгоритм верен и при любом количестве потоков вычисляет значения корректно.

```
tm5u7@login1:~/anty/C_Pthreads
$ ./C_pthreads_test 8
TEST #1 with parameters: 1000000 1 12.400000 23.100000 -107.580000
TIME: 1.726194
TEST #1 Result median difference: -0.067698
TEST #2 with parameters: 1000000 2 12.400000 23.100000 -107.580000
TIME: 2.408699
TEST #2 Result median difference: 0.014057
TEST #3 with parameters: 1000000 4 12.400000 23.100000 -107.580000
TIME: 2.466696
TEST #3 Result median difference: 0.003253
TEST #4 with parameters: 1000000 8 12.400000 23.100000 -107.580000
TIME: 2.292451
TEST #4 Result median difference: -0.007700
TEST #5 with parameters: 1000000 16 12.400000 23.100000 -107.580000
TIME: 2.367529
TEST #5 Result median difference: -0.007142
TEST #6 with parameters: 1000000 32 12.400000 23.100000 -107.580000
TIME: 2.086945
TEST #6 Result median difference: -0.018827
TEST #7 with parameters: 1000000 64 12.400000 23.100000 -107.580000
TIME: 2.007175
TEST #7 Result median difference: 0.008847
TEST #8 with parameters: 1000000 128 12.400000 23.100000 -107.580000
TIME: 2.019774
TEST #8 Result median difference: -0.003648
tm5u7@login1:~/anty/C_Pthreads
```

Рисунок 2 – Тест точности вычислений.

2.1 C & Linux pthreads

Библиотека *pthreads* использует подход, где разделяемая память предоставляет позволяет получать доступ к переменной результата с использованием примитива синхронизации – мьютекса.

Из библиотеки pthreads были взяты методы:

- *pthreadcreate* – функция, создающая поток.
- *pthreadjoin* – функция, принимающая данные от потока, который закончил свое выполнение.

Таким образом, программа создает необходимое количество потоков, после чего каждый из них считает свою часть значений функции от случайно выбранных на интервале точек. Главный поток программы дожидается, пока все потоки закончат свое выполнения, и вычисляет значение интеграла по общей формуле.

Преимуществом является использование простой и быстрой библиотеки на языке C, недостатком является наличие критической секции кода при

увеличении значения общего результата и как следствие - блокировки мьютексов *pthread_mutex_lock*, которые замедляют работу программы.

2.2 C & OpenMP

OpenMP (Open Multi-Processing) - это набор директив препроцессора, библиотек и переменных окружения, которые позволяют распараллеливать код на многопроцессорных системах с общей памятью. При этом разные процессоры будут иметь доступ к одной и той же ячейке памяти.

Процесс состоит из одного потока (главного), создающего несколько задач для каждого уровня рекурсии, а затем все потоки могут выполнять задачи из этой сгенерированной очереди задач.

#include<omp.h> — это заголовочный файл для openmp, чтобы использовать его функции.

#pragma omp parallel for reduction(+:global_sum) определяет параллельную область, цикл который будет выполняться, используя несколько параллельных потоков. Этот код будет разделен между всеми потоками.

Директива *reduction* указывает, какие операции должны выполняться с общими переменными после завершения цикла.

Как только все потоки закончили вычисления, результат суммируется в переменной *global_sum* и вычисляется по формуле.

2.3 C & MPI

MPI — это библиотека интерфейса передачи сообщений, позволяющая выполнять параллельные вычисления путем отправки кода на несколько процессоров.

Каждый процесс генерирует свой набор случайных точек и вычисляет сумму значений функции на этих точках. Затем значения сумм собираются с помощью функции *MPI_Reduce(),* которая вычисляет сумму всех

значений и передает ее процессу с рангом 0, где окончательное приближенное значение интеграла вычисляется и выводится на экран.

Также использовались следующие функции:

- *MPI_Comm_rank()* возвращает ранг текущего процесса
- *MPI_Comm_size()* возвращает общее количество процессов
- *MPI_Bcast()* передает значение переменной "n" от процесса с рангом 0 ко всем остальным процессам
- *MPI_Reduce()* выполняет операцию редукции, в данном случае суммирования значений, и передает результат процессу с рангом 0.

MPI_Finalize() завершает работу MPI.

2.4 Python & MPI

Для языка Python работа программы была простроена аналогичным образом. Модуль библиотеки Python *mpi4py* предоставляет привязки Python для стандарта MPI.

3 ТЕСТИРОВАНИЕ

Различные технологии были протестированы на одинаковом количестве узлов, процессов и потоков, но на разном количестве случайных точек.

В таблице представлены результаты тестирования для N=1000, 10000, 100000, 1000000, одного узла и двух процессов (потоков). Время работы программы представлено в секундах:

Рейтинг	Технология	1000	10000	100000	1000000
1	OpenMP\C	0.000006	0.000055	0.000419	0.002545
2	MPI\C	0.000409	0.002757	0.024694	0.164308
3	pthread\C	0.004897	0.041332	0.229795	2.645697
4	MPI\Python	0.001957	0.017024	0.168649	4.501786

По результатам, наилучшим образом справляется с задачей технология OpenMP, показывая наименьшее время работы программы. Стоит отметить, что pthread показали худший результат в сравнении с PythonMPI на небольших выборках. Что вероятнее всего связано с блокировками процессов на мьютексах и недостаточной оптимизацией параллельного выполнения программы. Однако при увеличении количества точек видно, что время работы MPI растет быстрее (особенно для Python), чем время работы программ, использующих многопоточность (pthreads, OpenMP) и при высоких значениях N получаем преимущество при использовании pthreads в сравнении с PythonMPI.

Далее каждая технология была протестирована для различных исходных параметров (количество процессов, потоков, узлов) и фиксированном количестве точек = 1000000.

Pthreads:

Количество потоков	Время, сек
2	2.408699
4	2.466696
8	2.292451
16	2.367529
32	2.086945
64	2.007175
128	2.019774

OpenMP:

Количество потоков	Время, сек
2	0.003038
4	0.004051
8	0.004282
16	0.004650
32	0.004217
64	0.004791
128	0.004566

C MPI:

Количество потоков	Время, сек
2	0.046994
4	0.023883
8	0.013995
16	0.010056
32	0.005172
64	0.005805
128	0.004466

Python MPI:

Количество потоков	**Время, сек**
2	1.6804590225219727
4	1.6880340576171875
8	1.910006046295166
16	3.428046464920044
32	3.76658296585083
64	7.433868408203125
128	11.808297634124756

Результаты тестирования MPI для различного количества узлов, время указано в секундах.

Количество узлов	**C_MPI**	**Py_MPI**
1	0.046994	1.6804590225219727
2	0.046892	1.6751270294189453
3	0.046842	1.677306890487671
4	0.025817	1.5250392436981201

ВЫВОД

В ходе лабораторной работы был разработан алгоритм вычисления определенного интеграла методом Монте-Карло, использующий параллельные вычисления. Алгоритм был реализован для следующих технологий:

- C & Linux pthread
- C & MPI
- Python & MPI
- C & OpenMP

Каждая программная реализация была протестирована для различных значений случайных точек, потоков, узлов, процессов.

Тест точности был пройден всеми разработанными программами.

Наилучшее значение производительности показала программа с использованием OpenMP, что связано с использованием компилируемого языка C и алгоритмическим распараллеливанием задачи, что исключает ошибку разработчика и неэффективное использование примитивов синхронизации.

В ходе эксперимента с изменением количества потоков выяснилось, что для технологий Pyhton MPI и OpenMPI производительность оптимальна при 2 потоках и падает при увеличении количества потоков. Для pthreads и C_MPI скорость вычислений увеличивается.

ПРИЛОЖЕНИЕ

Исходные коды реализаций размещены в репозитории:

https://github.com/anty-nor/SC_multithreading_Montecarlo_integration