Министерство образования и науки Российской Федерации

Новосибирский национальный исследовательский государственный университет

Основы параллельного программирования

Отчет по лабораторной работе № 2

Студент: Матеюк Илья Анатольевич (aka qwerty123md)
Преподаватель: Сарычев Виктор Геннадьевич

Новосибирск, 2020 г.

1. ***Цель работы***

Разработать и исследовать параллельные программы решения СЛАУ методом сопряженных градиентов с применением библиотеки OpenMP

2. ***<u>Краткое</u> описание подходов к организации решения прикладной задачи параллельными взаимодействующими процессами***

Реализованы 2 подхода к организации параллельной программы при умножении матрицы на вектор:

1) *Для каждого распараллеливаемого цикла создается отдельная параллельная секция #pragma omp parallel for*
2) *Создается одна параллельная секция #pragma omp parallel, охватывающая весь итерационный алгоритм*

Для каждого подхода при профилировании писались логи в log.txt, который Вы сможете найти в каждой из папок stat<N>, где N = {0,1,2} (без оптимизаций, 1 вариант, 2 вариант) Все графики построены на основании данных из логов.

3. ***Профилирование***

С помощью time было установлено, что для работы программы на протяжении > 30 sec потребуется симметричная матрица 1000 x 1000, и ε = 1E - 25
Компилируем и используем gprof
gcc main.c cgm.c cgm.h -lm -o go -pg

time	seconds	seconds	calls	s/call	s/call	name
98.82	29.53	29.53	8240	0.00	0.00	mul_matrix_vector
0.33	29.63	0.10	32956	0.00	0.00	scalar_mul
0.23	29.79	0.07	24717	0.00	0.00	mul_num_vector
0.17	29.91	0.05	16478	0.00	0.00	add_vectors
0.10	29.95	0.04	8240	0.00	0.00	check_end_alg
0.10	29.98	0.03	8240	0.00	0.00	sub_vectors
0.07	29.99	0.02	1	0.02	0.02	initial_alg
0.00	29.99	0.00	1	0.00	29.99	do_magic

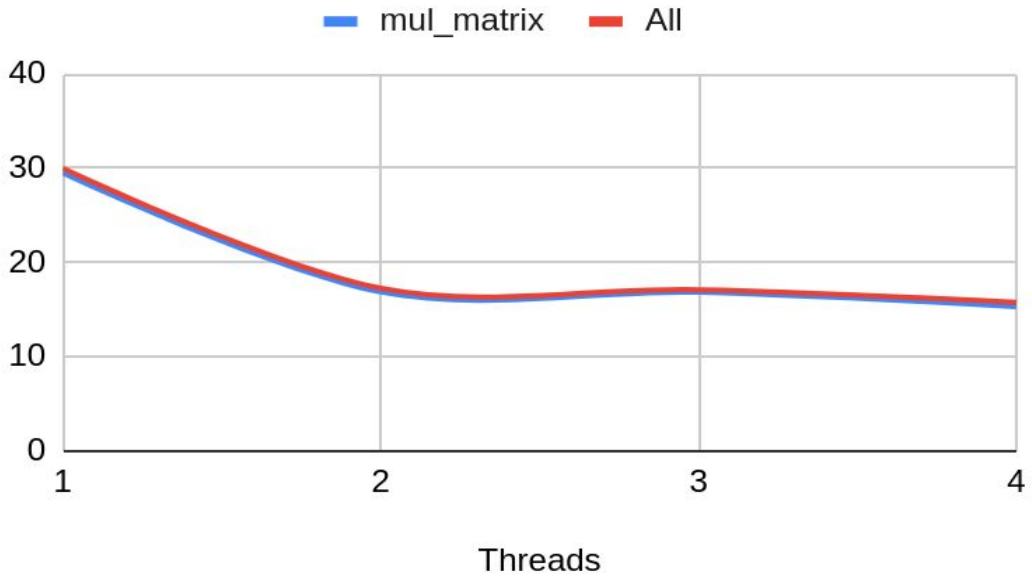
Основную часть времени (98.9 %) занимает умножение матрицы на вектор. Также часто вызывается scalar_mul и mul_num_vector (33к и 25к), но пытаясь их распараллелить, я пришел к выводу, что оптимизировать функции, занимающие менее секунды процессорного времени, смысла нет - внутренние процессы OMP занимают больше времени, чем мы выигрываем.

4. *Исследование производительности программ*

Для всех замеров использовалась конструкция:

```
start = omp_get_wtime();
//do_smth
end = omp_get_wtime();
timers.<> += end - start;
```

Вариант 1:
Использовав директиву `#pragma omp parallel for` *и изменяя* `num_threads()`*, был получен следующий график.*

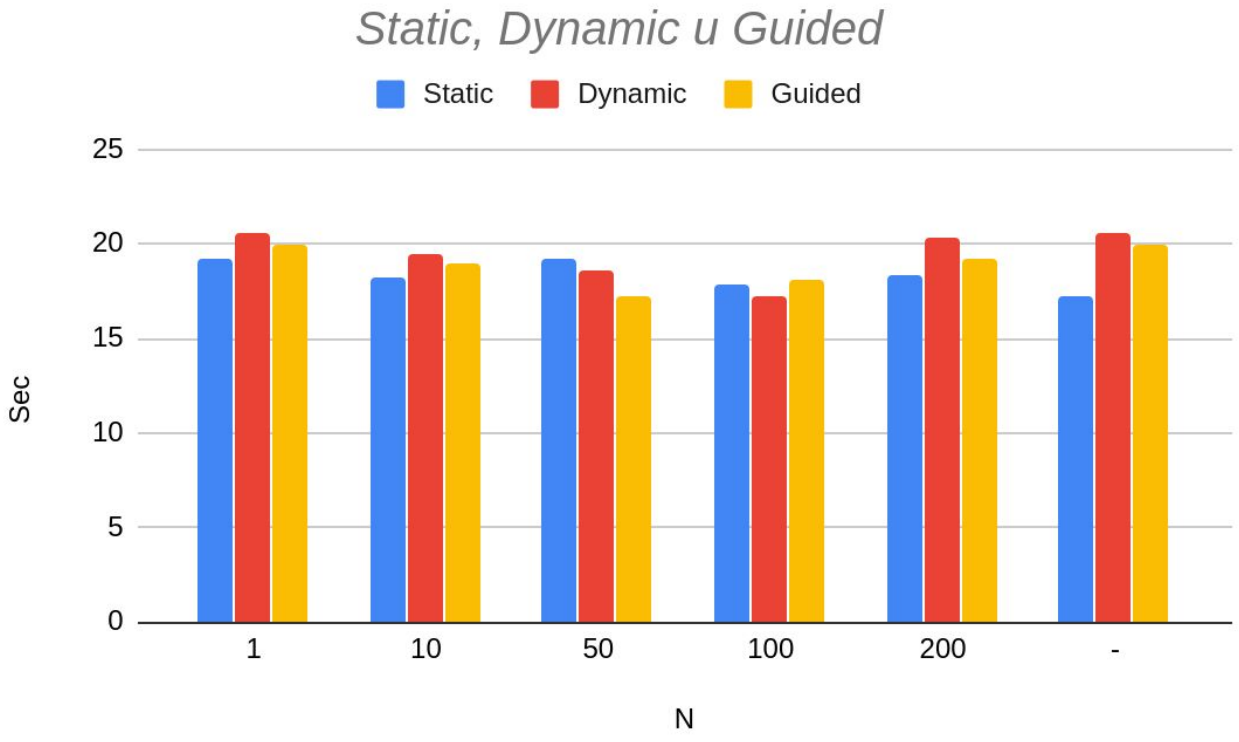


Отсутствие прироста на 3 и 4 потоках обусловлена наличием технологии hyper-threading. Дальнейшие выкладки будут для 1 и 2 потоков.
(в log.txt Вы сможете посмотреть полные логи)

Далее следовало узнать наиболее оптимальный из параметров schedule (static, dynamic, quided).

- *static - все итерации цикла поровну делятся между потоками. Часто дает наилучший результат.*
- *dynamic - каждый поток получает заданное число итераций, выполняет их и запрашивает новую порцию*
- *quided - распределение начинается с некоторого начального размера, зависящего от реализации библиотеки до N. Размер выделяемой порции зависит от количества еще нераспределенных итераций*

Рассмотрим график, чтобы сделать вывод о целесообразности использования того или иного параметра:



Как можно заметить, наиболее эффективна наша программа, если использовать static без параметров. Это легко объяснить - в нашей программе каждое действие в цикле равноправно и выигрыша при использовании dynamic или guided не будет. Но если бы наша матрица была сильно разряжена или скажем, нужно было вычислять сумму элементов под главной диагональю, то результат замеров был бы совершенно иной.

Проведем итоговое сравнение двух вариантов реализации:

Используем *num_threads(4)* и *schedule(static)*

	Var 1			Var 2	
Threads	1	2		1	2
Time	29,889	15,69		29,889	17,95
Acceleration	1	1,90		1	1,66
Parallel coef	1	0,95		1	0,83

5. *Заключение*

Реализованы два варианта многопоточной программы решения СЛАУ. Эффективность обоих высока, но первый вариант выигрывает по времени, так как распараллеливание имело более частный подход, прибегающий к распараллеливанию частей кода, наиболее требовательных к ресурсам процессора. Исход сравнения был заранее спрогнозирован средствами профилирования.
Таким образом, следует наиболее внимательно выбирать код, нуждающийся в оптимизации и отдавать предпочтение его точечному распараллеливанию.
Вывод: при правильном применении библиотеки OpenMP можно наиболее эффективно использовать ресурсы ЭВМ и тем самым уменьшить время выполнение задачи.