Министерство образования и науки Российской Федерации

Новосибирский национальный исследовательский государственный университет

Основы параллельного программирования

Отчет по лабораторной работе № 1

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***

После написания программы я задался вопросом - а что распараллеливать - то? Дабы это разузнать было принято решение прибегнуть, как вы уже догадались, к профилированию.
Для начала следовало определиться с начальными параметрами. В моей программе - это конфигурационный файл, содержащий размер матрицы, матрицу и вектор результата. Пошаманив с time я выяснил, что для работы программы на протяжении > 30sec потребуется симметричная матрица 1000 x 1000, которая в итоге была успешна нагенерена в входной файл.
Далее нужно было узнать, какая часть моей программы самая затратная:
компилируем gcc main.c cgm.c cgm.h -lm -lrt -o go -pg
и даем gprof'у распарсить получившийся gmon.out

```
qwerty123@asus:~/parallel-programming/lab_1/src$ gprof ./go -b
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 98.92     29.70    29.70     8240     0.00     0.00  mul_matrix_vector
  0.50     29.85     0.15    32956     0.00     0.00  scalar_mul
  0.40     29.97     0.12    24717     0.00     0.00  mul_num_vector
  0.17     30.02     0.05    16478     0.00     0.00  add_vectors
  0.07     30.04     0.02     8240     0.00     0.00  sub_vectors
  0.03     30.05     0.01     8240     0.00     0.00  check_end_alg
  0.03     30.06     0.01        1     0.01    30.06  do_magic
  0.00     30.06     0.00        1     0.00     0.00  free_data
  0.00     30.06     0.00        1     0.00     0.00  initial_alg
  0.00     30.06     0.00        1     0.00     0.00  print_solution
```

Как мы видим, основную часть времени (98.9 %) занимает умножение матрицы на вектор. Так же довольно много раз вызывается scalar_mul и mul_num_vector (33к и 25к), ради эксперимента их я тоже пытался распараллелить, но в итоге пришел к выводу, что оптимизировать функции, занимающие менее секунды процессорного времени, смысла нет - внутренние процессы OMP занимают больше времени, чем мы выигрываем.

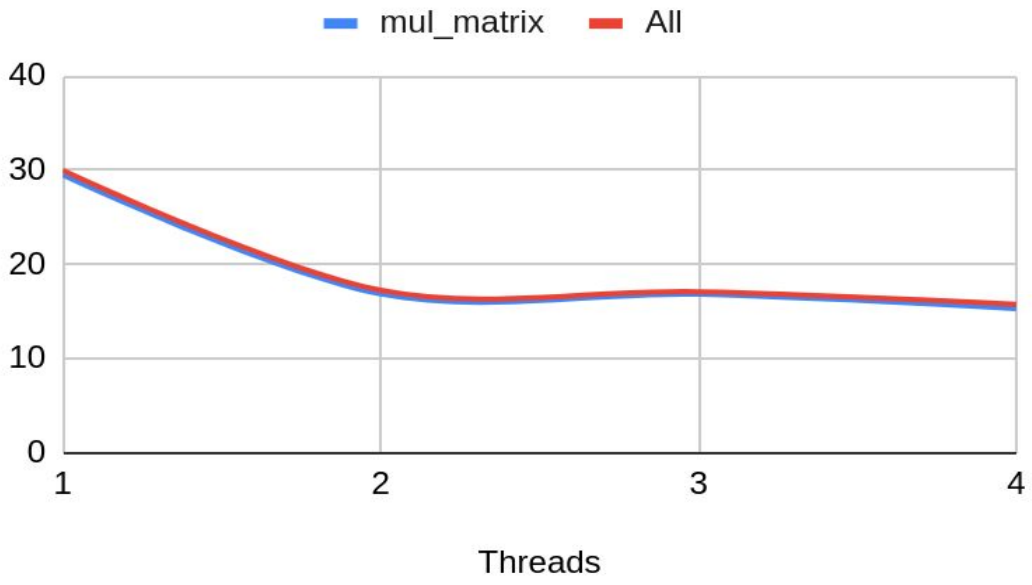
4. *Исследование производительности программ*

Для всех замеров использовалась конструкция:

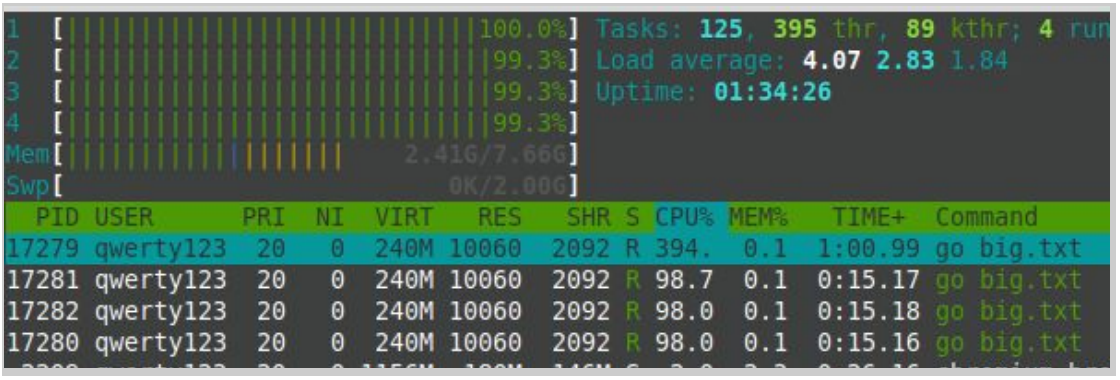
```
clock_gettime(CLOCK_REALTIME, &start);
//do_smth
clock_gettime(CLOCK_REALTIME, &end);
timers.<> += 1000000000 * (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec);
```

Вариант 1:

Использовав директиву `#pragma omp parallel for` *и поиграв с параметрами* `num_threads()`*, я получил следующий график.*

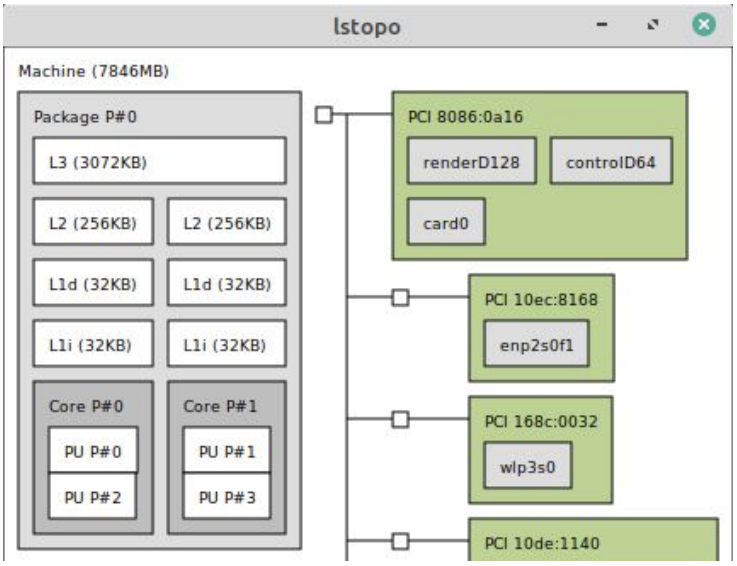


Сначала меня это смутило, ведь `cat /proc/cpuinfo` *любезно сообщил нам о том, что на моей машине 4 ядра (а я еще и не вчитывался в core_id)! Более того все 4 трудились в поте лица, при установке* `num_threads(4)`



Но как так, ядер на машине 4, а ускорение мы получаем только в 2 раза, меня что обманули !?

Когда я визуализировал архитектуру процессора, все стало на свои места:



Ядер у нас не 4, а всего 2. Но зато каждое из них имеет два логических ядра. Это технология называется hyper-threading. Суть в том, что АЛУ у нас по прежнему один, но регистры и APIC дублируются. К сожалению, в нашей задаче особого прироста эта технология нам не даст, поэтому дальнейшие выкладки будут для 1 и 2 потоков, но при желании в log.txt Вы можете просмотреть ВСЕ подробные логи.

//зануда mode off

Теперь можно перейти к сути:

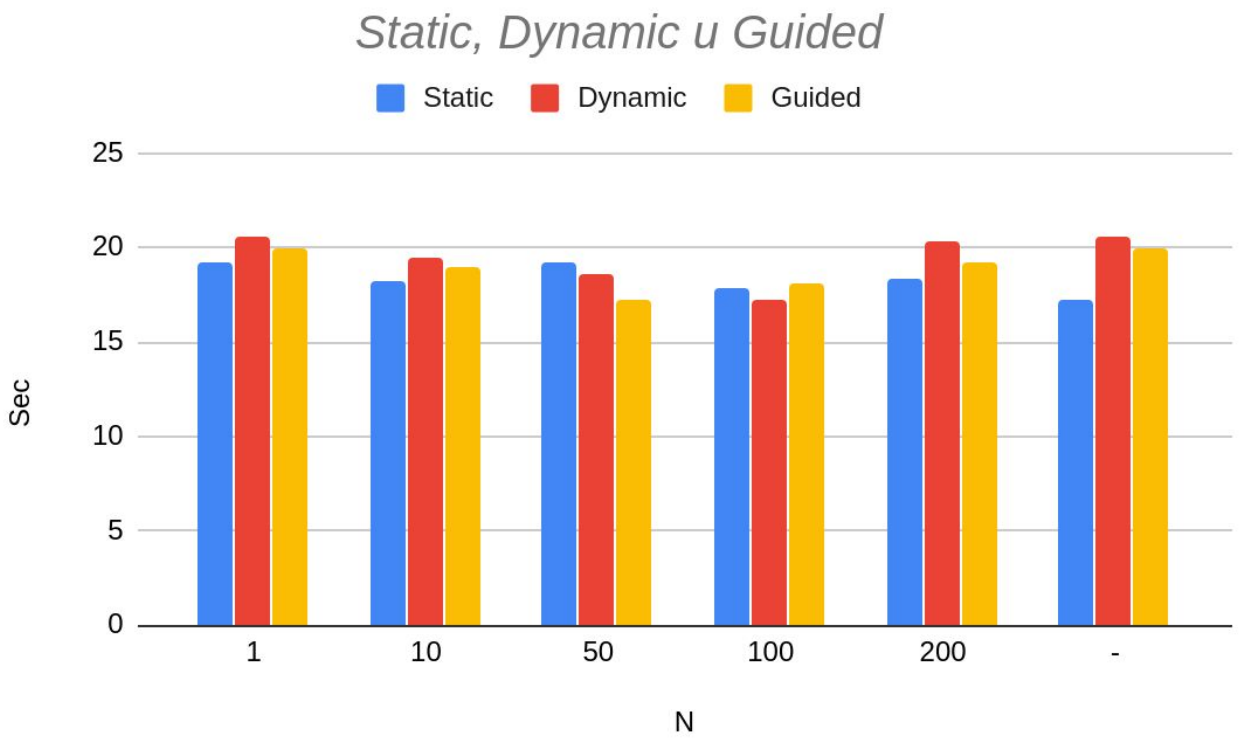
Если вопрос с тем, на сколько потоков распараллеливать задачу отпал, то какой из параметров schedule (static, dynamic, quided) использовать - все еще остался.

- *static - некое олицетворение коммунизма - 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 =)

Теперь самое время посмотреть на то, ради чего все это затевалось:

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

Можем заметить, что создание одной параллельной секции проигрывает точечному распараллеливанию частей кода, наиболее требовательных к ресурсам процессора.
Таким образом, следует наиболее внимательно выбирать код, нуждающийся в оптимизации и отдавать предпочтение его частному распараллеливанию.

5. Заключение

Самое главное, что я уяснил выполняя лабораторную работу - это то, что не следует бездумно пытаться распараллелить все, где есть циклы. Я выяснил, что наиболее разумный подход - это избегание преждевременной оптимизации, повсеместное использование инструментария для профилирования и тщательный анализ параметров оптимизации.
Сравнив два предложенных способа реализации программы, я сделал выводы о целесообразности этих подходов.
Помимо всего вышеизложенного, мне удалось поработать со многими другими средствами библиотеки OpenMP (выделение критических секций, установка барьеров и регионов выполнения, использование shared, private, reduction переменных и т.д), но все они выходят за рамки этой работы.