**Problem 1:**

Gprof- I added –pg compiler flag to Makefile and everything worked beautifully. When run the program created an extra output gmon.out, which could be run at command line with gprof gmon.out profiledata.txt to have gprof examine the .out and translate it to readable file I named profiledata.txt.

The profiledata.txt contained relevant info such as:

%time spent in each routine, and the time spent in each routine

For the purposes of this assignment, I saw that the method which added columns first and then rows took 420.65 ms, where as adding rows first and then columns took only 40.06 ms.

PAPI- Significantly more involved. I had to load the module, run some mystery command to correctly allow performance profiling, and copy papi.h, papiStdEventDefs.h, and libpapi.a to the folder containing my homework code. I was able to determine the directory containing these files with the find command at terminal. Terminal commands looked like the following when run in my homework folder:

module load papi

/var/lib/pcp/pmdas/perfevent/perfalloc &

cp /util/academic/papi/5.4.1/include/papi.h ..

cp /util/academic/papi/5.4.1/include/papiStdEventDefs.h ..

cp /util/academic/papi/5.4.1/lib/libpapi.a ..

Where the last three commands only had to be run one time and needed complementary calls in the Makefile and hw2p1papi.cpp, whereas the former two were standalone and required at every login to the CCR.

We needed to collect data about the number of floating point operations, their speed, and the number of cache misses. PAPI functions needed to be called for each of these pieces of information to be collected. I chose to implement these functions one at a time for each of the two matrix addition methods. The results are as follows:

|  |  |  |
| --- | --- | --- |
| Data Type | Rows First | Columns First |
| MFlop/s | 33.1017 | 13.3011 |
| L1 Cache Misses | 9637877 | 126423288 |
| L2 Cache Misses | 8493141 | 83057932 |
| L3 Cache Misses | 9290022 | 9923575 |

So we can see that adding columns first has a significantly smaller MFlop/s, which is likely due to the extra time required to get the correct data in the L1/L2 caches before the flop can be performed.

**Problem 2:**

Code was written to calculate the area of the Mandlebrot set. Performance data was collected with Gprof using a serial run, parallel with dynamic scheduling, and parallel with static scheduling:

|  |  |  |  |
| --- | --- | --- | --- |
| Data Type | Serial | Parallel Dynamic | Parallel Static |
| Time | D = 223.568s  (S=210.904s) | 112.566s | 108.735s |
| Area | 1.50702 | 1.50702 | 1.50703 |
| Num. Processors | 1 | 4 | 4 |
| Speed Up | - | 198.61% | 193.961% |
| Serial Fraction | - | 33.7997% | 35.4091% |

Somewhat unexpectedly the Static scheduling has very close results to dynamic, even though there would be a large number of initial values used by the threads which would immediately diverge in the Mandlebrot testing function. I suspect this is due to my using the laptop for other things while running the dynamic version of the code, as can be seen by the serial run time for the Dynamic data collection (D), versus the serial run time for the Static data collection (S).

As for weak and strong scalability:

Strong scaling can be calculated simply as the speed up achieved in processing data of a set size as the number of processors is increased, so we can just run the program with

1 <= p <= max\_threads and see how the performance changes with increasing threads.

Weak scaling would be how much data can be processed by a number of threads if the maximum time is fixed, so we need to change the code to break when a time limit is reached, again for multiple numbers of threads and print out the number of points checked for each value of p.

We can naively estimate these values with the following equations:

S\_strong <= T\_serial/T\_parallel = Speed Up

S\_weak <= p – F (p-1)

Where F is the serial fraction when the work per processor is fixed. I will estimate F to be equal to the serial fraction as calculated from Karp-Flatt when the scheduling is set to static.

The strong scaling with p = 4 was thus <= 1.98.

The weak scaling with p = 4 is then <= 2.94.

We would expect a higher weak scaling for this problem since more processors would directly imply a larger number of points checked in a set time frame. This is in contrast to the entire data set being checked by an increasing number of threads, which would yield a speed up, but introduce other factors such as the overhead of creating/destroying the threads.

However, a more detailed investigation with varied number of processors would need to be performed to validate this estimation. I had some difficulties with implementing such a study, especially with setting a time limit for weak scaling.

**Problem 3:**

We are to check the Goldbach conjecture up to some large even integer ~10^5.

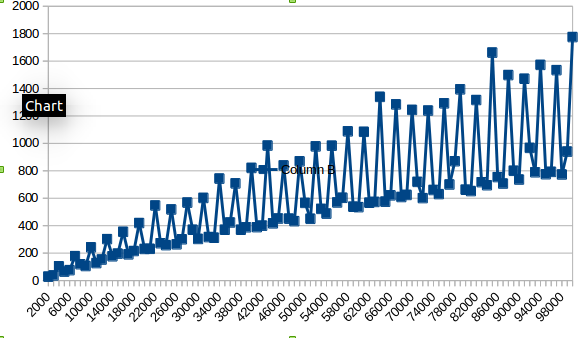
When checking up to 10^4, the following times and serial fractions were found:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data Type | Serial Bool | Serial Count | Par. Outer Bool | Par. Outer Count | Par. Inner  Bool | Par. Inner  Count |
| Time (s) | 0.0165342 | 0.618057 | 0.0043506 | 0.258067 | 0.0135696 | 0.305355 |
| Serial Fraction | - | - | 1.75026% | 22.3393% | 76.0931% | 32.5409% |

When checking up to 10^5, the following times and serial fractions were found:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data Type | Serial Bool | Serial Count | Par. Outer Bool | Par. Outer Count | Par. Inner  Bool | Par. Inner  Count |
| Time (s) | 0.281559 | 121.764 | 0.124146 | 63.2666 | 0.469522 | 69.5704 |
| Strong Scaling Est. | - | - | 2.268 | 1.925 | 0.5997 | 1.750 |
| Weak Scaling Est. | - | - | 3.236 | 2.917 | -1.670 | 2.715 |
| Serial Fraction | - | - | 25.4567% | 35.9444% | 189.011% | 42.8471% |

We can see that the overhead is too high for the inner loop to be parallelized, since the calculations are so fast, particularly for the boolean check.

The theorem seems to hold since the number of prime pairs which sum to a given even number trends upward as the even number increases. This is not to say that n+2 has more prime pairs than n, but generally there are more prime pairs for n+10000 than for n. This can be seen in the following graph of prime pairs vs even number:

We may investigate weak and strong scaling by the same method as problem 2, the naive estimations of which for p = 4 are in the table above.

We again see that the problem is more weakly scalable, especially for the outer loop. It is easy to imagine that if each thread is checking one even number, that an increase in threads leads to an increase in numbers checked in a set time. However, the overhead becomes a larger factor for the inner loop, where the simple boolean check makes parallelization a huge detriment.

When the maximum run time is set to 60 seconds, data to study weak scaling can be generated by varying the number of threads and counting the number of pairs checked. I wasn't able to implement this in time however...

**Problem 4:**

We are to perform numerical integration to estimate the value of pi with a variety of parallelization methods. Performance data is displayed below:

|  |  |  |  |
| --- | --- | --- | --- |
| Data Type | Serial | Static Parallel | Dynamic Parallel |
| Time (s) | 15.3573 | 29.4551 | 42.3887 |
| Speed Up | - | 36.2298% | 52.138% |
| Serial Fraction | - | 334.688% | 222.398% |

We can see that the Speed Up is below 100%, so it actually took longer to compute pi with multiple threads, which is also reflected in the serial fraction above 100%. This is due to the overhead of creating and destroying threads, since the calculation each thread is performing is minuscule in this problem.

Weak and strong scaling are studied again as outlined in problem 2.

For p = 4 the calculated values are:

Strong Scaling <= .52138

Weak Scaling <= -6.04064

We can see that this problem is not scalable, since each thread is created, performing just one calculation, passing that information back to root, and then being recycled.

A deeper investigation of strong scaling shows the speed up for num\_threads from 1 to 4 is as follows:

Static: 1.00 → 0.41 → 0.57 → 0.36

Dynamic: 1.00 → 0.21 → 0.28 → 0.52

So we can see that when the scheduling is static the time that each thread is waiting increases to the detriment of efficiency, but when scheduling is dynamic there is an improvement in efficiency as the number of threads increase, but serial processing was still superior.

There is again however, large variability due to the data being collected on my laptop, which I was using to write this report during the calculations. A study with a higher range of processors should have been performed on the CCR to get accurate results for the weak and strong scaling of problems 2, 3 and 4.