OpenMP
Assignment #3, CSC 746, Fall 2021

Arulselvan Madhavan*
SFSU

ABSTRACT

In this paper, I present the results of studying the speedup achieved by parallelizing a program across a varying number of cores. The objective of this experiment was to study the difference between theoretical speedup and the actual speedup achieved from distributing the tasks across cores. A simple Matrix Vector product was selected for the experiment. OpenMP was enabled and used to parallelize the program across the cores. Serial version of CBLAS(C implementation of Basic Linear Algrebra Subprograms) was used to validate the results of the implementation.

1 INTRODUCTION

With the increase in number of cores and the ever increasing amount of data to process, there is a need to utilize the machine resources to the maximum. Theoretically, taking a serial program and running it in parallel is expected to give a T(n)/p speedup, where T(n) is the runtime of the serial program and p is the amount of parallelization added. But, in reality, this speedup is not seen, for most programs. The aim of this experiment is to understand how the speedup achieved scales from increasing the amount of parallelization(p).

Matrix-Vector multiplication is a common linear algebra operations which lends itself well for parallelization. A serial Matrix vector multiplication was implemented and validated against CBLAS package. The Matrix vector multiplication was then parallelized using OpenMP pragmas as presented in the $2. The threads used by openMP was controlled via command line and so was the thread scheduling pattern used by OpenMP. The thread counts tried were 1,2,4,8,16,32,64,272. Static and dynamic scheduling pattern were tried.

The results show that the parallelized version of matrix-vector outperforms the serial CBLAS, as expected, by several orders of magnitude. Speedup achieved from parallelization was always below the theoretical peak as expected. The maximum speedup achieved was around 115, when 272 threads were used for parallelization and static scheduling pattern was chosen. The main takeaway is that it's worth parallelizing our programs as long as the cost that comes with it doesn't outweigh the speedup gained from parallelizing the program.

2 IMPLEMENTATION

The first set of experiments compared the serial implementation of matrix-vector product(y[j] = y[j] + A[i,j] * x[j]) with serial CBLAS implementation.

2.1 Part 1

The objective of this setup is to get a baseline estimate of the execution times that come with running a serial matrix-vector product for a variety of problem sizes(1024,2048,4096,8192,16384). CBLAS implementation was used to validate the results.

*email:amadhavan@sfsu.edu

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    y[i] = y[i] + A[i*n + j] * x[j];
  }
}
```

Listing 1: Serial Matrix vector product implementation

2.2 Part 2

The aim of this setup is to parallelize the serial implementation with simple openmp pragmas. OpenMP provides a setup of compiler pragmas that allows us to parallelize the serial programs. Since the matrix vector product is a simple 2-level nested for loop, the outer loop can be parallelized and run on different threads. The below implementation shows the pragmas used for parallelizing the matrix-vector product. The parallelization is limited to the outer for loop. The variables private to the threads are the loop variables i and j. All other variables are shared between the threads.

```
/*
 * This routine performs a dgemv operation
 * Y :=  A * X + Y
 * where A is n-by-n matrix stored in row-major
    format, and X and Y are n by 1
 * vectors. On exit, A and X maintain their input
    values.
 */

void my_dgemv(int n, double *A, double *x, double
    *y) {
  int i, j;
#pragma omp parallel for default(none) shared(n, A
    , x, y) private(i, j)
  for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
      y[i] += A[i * n + j] * x[j];
    }
  }
}
```

Listing 2: Parallel Matrix vector product implementation

3 EVALUATION

This section contains the results from running the aforementioned implementations for the following problem sizes(n). A n*n matrix(A) was considered as the input matrix. A n-length vector(x) was taken as the input vector. A n-length vector(y) was used as the output vector. Both implementations were run for problem sizes - 1024,2048,4096,8192,16382. The parallel implementation was run on threads - 1,2,4,8,16,32,64,272. The threads were scheduled using static and dynamic scheduling, and results were collected separately. Both implementations were run on a Cori KNL node.

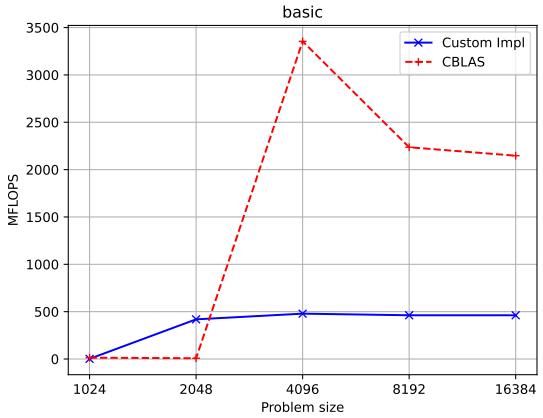

Figure 1: Serial Matrix-vector product custom implementation vs CBLAS

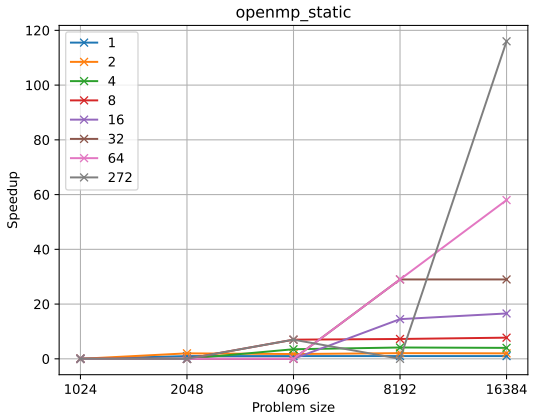

Figure 2: Parallel Matrix-vector product with static scheduling vs CBLAS

3.1 Computational platform and Software Environment

The specifications were taken from `https://docs.nersc.gov/systems/cori/`

- CPU: Intel's Knights Landing Processor
- CPU clock cycle: 1.4GHZ
- Memory Bandwidth: 102 GiB per sec
- Theoretical Peak FLOPS per node: 29 PFlops
- Number of cores: 68
- CPU per core: 4
- Maximum number of threads: 272
- L1 caches: 64 KB (32 KiB instruction cache, 32 KB data)
- L2 cache: 1MB
- Compiler: gcc 8.3.0 (O3 optimization)
- OS: SUSE Linux Enterprise Server 15
- OpenMP version: 201511

3.2 Methodology

3.3 Experiment 1

The implementations were instrumented with chrono timer library to collect the execution time of the program. From the execution time, Flops(Floating point operations per second) was calculated using the formula 2 * n * n, where n is the problem size. This metric was used to compare the serial implementation 2.1 and serial CBLAS. The results are presented in Figure:1. Serial CBLAS outperforms the serial implementation from 2.1. Maximum MFlops achieved was 500.

3.4 Experiment 2

The implementations were instrumented with chrono timer library to collect the execution time of the parallelized program from 2.2. The number of threads used by the parallel implementation and the scheduling strategy used were controlled using environment variables. The implementation was tried for thread counts - 1,2,4,8,16,32,64,272. Static and dynamic scheduling pattern were

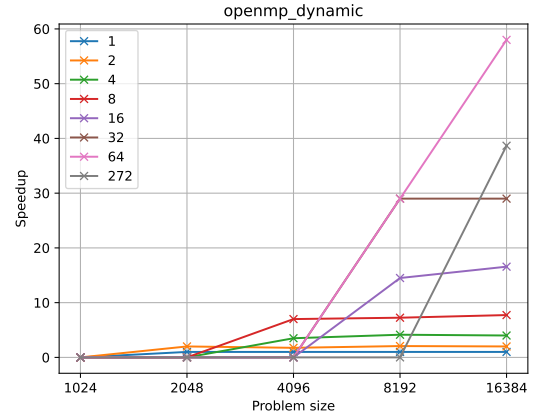

Figure 3: Parallel Matrix-vector product with dynamic scheduling vs CBLAS

also tried. The objective of this experiment is to calculate speedup. Speedup was calculated using the formula - execution time of the serial code/ execution time of the parallel code. Expected Theoretical peak of the parallelized program was calculated using the formula - execution time of serial code / number of parallel threads. The speedup achieved from parallelizing the problem across threads - 1,2,4,8,16,32,64,272 using static scheduling is given in Figure:2. The results from dynamically scheduling the threads are shown in Figure:3.

3.5 Findings and Discussion

- Results from Figure:1 show that the serial implementation of matrix vector product performed almost four times worse than the serial CBLAS implementation.
- The MFLOPS achievable from the serial implementation averaged around 500.
- Serial CBLAS showed interesting bump in MFLOPS for problem size 4096. The bump in for this problem size could be from being able to fit the data needed for the computation in L1 cache.

- Serial CBLAS also performed worse than my implementation. I think this is because for smaller problem sizes the overhead that comes with optimized code outweigh the benefits and hence resulted in worse performance numbers for smaller problem sizes - 1024 and 2048.
- Overall, serial CBLAS convincingly outperformed my implementation.
- Matrix vector parallelized over threads 1,2,4,8,16,32,64,272 showed increase in speedup as the number of threads that were made available to perform the computation were increased.
- The speedup achieved from parallelizing, at times, came close to the theoretical peak, but didn't achieve it. The best speedup was from statically scheduled 272 threads. The best performing configuration vs serial CBLAS is shown in Figure:4
- Cori's KNL nodes have 68 cores and 4 CPUs per core. So, a total of 272 cores are available for parallelization. So, max threads chosen for this experiment was set to 272 cores.
- As we can see from Figure:4, the max achieved FLOPS from the parallelized program is close to 50 GFlops whereas the serial CBLAS is 3 GFlops.
- This comparison between parallelized implementation and serial CBLAS is not fair since CBLAS was not configured to run its code using multiple cores. The environment variables that control OpenMP threads were explicitly cleared while collecting the metrics for CBLAS implementation. So, openmp limited to run on a single core, at best can achieve a theoretical peak of 44.5 GFlops which is far lower than the theoretical peak of 29 PFlops that becomes possible when we make the other CPU cores available to it.
- A serial CBLAS at best can use the vector registers(AVX) to run the code but this is limited since the compiler conservatively chooses to use the vector registers when it has to share data between iterations.
- There is a lot of similarities between the statically scheduled OpenMP threads and dynamically scheduled OpenMP threads.
- Serial scheduling achieved a speedup close to 120 when run on 272 threads whereas the dynamically scheduled version only managed to achieve speedup close to 60, and this was achieved when run with 64 threads. Interestingly, the dynamically scheduling of 272 threads performed worse.
- The other speedups for thread counts 1,2,4,8,16,32 performed similarly between the statically scheduled and dynamically scheduled versions.
- Comparing Figure:2, Figure:2 and Figure:4, I can say that the openmp versions are not seeing a linear speedup with the increase in number of threads. This is evident in thread count 64 and 272. The speedup achieved close to 58 with 64 threads but only 120 with 272 threads(See Figure:2)
- The reason for not achieving linear speedup can be explained using Amdahl's law. S(n,p) = 1/(f + (1 - f)/p), where n is the problem size, n is the serial portion of the code, p is the parallelization factor. For any parallel program, some work has to be done serially to coordinate tasks between parallel cores, and collecting results. This contributes to a non-zero value to f which limits the peak achievable speedup.
- As p increases, the time spent in coordinating between tasks increases which limits the amount of speedup one can achieve by increasing p.

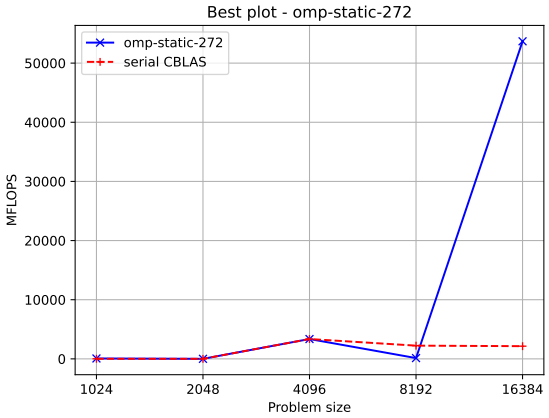

Figure 4: Parallel Matrix-vector product (best) vs serial CBLAS

REFERENCES

[1] Cori specifications - https://docs.nersc.gov/systems/cori/