Yaseen Aniss

Gaussian Elimination with OpenMP

Threads provide an efficient way to share memory yet utilize multiple units of execution. Using threads in this manner provides us with an efficient way to get closer to achieving true parallelism. The MTL computers contain multiple cores that share the same memory and by making sure that the number of threads is equal to the number of cores, each thread is allowed to operate on its own core, and is a freely schedulable entity. This allows multiple threads to run on multiple cores at the same time, which drastically improves throughput. In order to improve the speed of a parallelized program, one needs to account for many different factors, including but not limited to, the layout of your data, how your choose to distribute the work and partition the data in a parallelized region, and where/when to synchronize the work. In this paper, we will talk about my choices for these factors and my justifications for my choices.

When choosing how to layout my data, I was presented with two options:

1.) A contiguous layout for the array.
2.) Represent the array as a vector of n pointers to n-element data vectors.

I elected to choose option 2, which was to represent the array as a vector of n pointers to n-element data vectors. The reason I chose to layout my data this way was because I knew that we may need to execute n-1 row swaps at worst case during the pivot phase and I knew that if we were to represent the data as a contiguous layout, then these swaps would be costly. On the other hand, representing it with pointers allows us to simply exchange pointer value. This gives us the advantage of having our swap operations finish in constant time.

When choosing how to partition my data, I was presented with 5 options:

1.) Block partitioning.
2.) Static cyclic with large chunks
3.) Static cyclic with small chunks
4.) Dynamic
5.) Guided

To test which option would be the best for my program, I executed 30 runs with different chunk sizes and different scheduling types, but with the same matrix and vector data. I tested the 5 scheduling types against a problem size of n = 4000 with 5 threads, 10 threads, and 20 threads on Jaguar. I examined the performance of the following 10 options:

-Block
-Static with n/(num_threads * 2) size chunks
-Static with n/(num_threads) size chunks
-Static with n/(num_threads * num_threads) size chunks
-Static with the size of the chunks as 1
-Dynamic with n/(num_threads * 2) size chunks

-Dynamic with n/(num_threads) size chunks
-Dynamic with n/(num_threads * num_threads) size chunks
-Dynamic with the size of the chunks as 1
-Guided

The outcomes showed that scheduling with static partitioning using "n/(num_threads * 2)" size chunks provided the best results, therefore, that was the scheduling choice I used in my program. I believe one of the main reasons that static showed the best performance was due to the simplicity in overhead for implementing static cyclic scheduling compared to dynamic and guided scheduling.

When writing my code, I already knew the thread count I would use for each run (1, 2, 5, 10, 20, 30) and I wanted to make sure I accounted for load balancing inconsistencies. I realized that the formula "n/(num_threads * 2)" gave a 0 remainder for each potential thread number listed above, except 30 (i.e remainder of n/(30 * 2)! = 0). I accounted for this in my code by creating a function (calculate_chunk_size()) that finds the next best chunk size if n%(num_threads * 2) != 0. This guarantees that no matter the input, each thread will have the exact same amount of iterations distributed to it, which will improve performance since load balancing is accounted for.

Since I chose to use static scheduling in each parallelized section of my code, then the way the program utilizes the threads and distributes the work is by dividing the for loop iterations into equal-sized chunks before the loop begins executing. Each thread then works on its own to complete its designated set of work (i.e the iterations of the for loop it was assigned).

My program takes advantage of parallelism wherever possible, except for the initialization of the matrices/vectors and when calculating which row contains the largest absolute value during the pivoting phase. The reason I did not choose to partition the work in these two cases was because

1.) We were asked not to time the initialization of the matrices and vectors, so I did not feel the need to parallelize this portion of the program.
2.) Calculating the largest value in a column using multiple threads would take a lot of overhead and seemed like too much work. Also, parallelizing this portion of the program did not seem like it would reap much benefit due to all the communication that would be needed between the threads.

My program does, however, take advantage of parallelism in the following phases of Gaussian elimination:

Forward elimination:

In the forward elimination phase, each thread is given its designated iterations and rows to operate on. It then calculates the multiplier for the current iteration and uses this value to update the values in the current row. Once this is done, it also updates the B vector using the same multiplier and the same row number.

Back substitution:

During the back substitution phase, each thread is given its designated iterations and rows to operate on. It then proceeds to multiply its current A matrix value by the newly calculated X value and then take this result and subtract it from the corresponding B vector value.

Multiply A matrix by X vector:

During this phase, each thread is given its designated iterations and rows to operate on. It then calls a function during the parallel region called "cross_product," which is also parallelized.

Cross product:

During this phase, each thread is given its designated iterations and cross product portions to calculate. Once the threads calculate their individual portions of the cross product, they perform a "+" reduction using the OpenMP reduction clause, and then return the result to the calling function (the function above).

Subtract Vectors:

During this phase, each thread is given its designated iterations and subtraction result to calculate. The result of this calculation is passed to the "l2_norm" function, which then utilizes parallelization to calculate the l2-norm.

l2 norm:

During this phase, each thread is given its designated iterations and vector values to square and then perform a "+" reduction on the result using the OpenMP reduction clause.

Throughout my program, I only use the implicit OpenMP synchronization mechanisms. I did research on whether or not I should try to add some synchronized areas in my code, and I came to the undisputed conclusion that performance is at its most optimal speed when synchronization is minimized. After coming to this realization, I made it my goal to write my code in a way that avoided the need to use any explicit synchronization mechanisms. Therefore, the only synchronization points in my program are the implicit ones that OpenMP provides by default (the ones at the beginning and at the end of parallel constructs).

When examining the results of the runs on 1, 2, 5, 10, 20 and 30 threads/cores, I noticed that the efficiency decreased as the number of threads/cores increased. Also, the speedUp values increased at a smaller rate as the number of threads/cores increased. This suggests that the problem is weakly scalable and needs an increase in data size to keep the efficiency around the same value as more threads/cores are added. Some potential reasons for the decrease in efficiency are as follows:

1.) The chunk sizes get smaller as the number of threads/cores increases. This results in more overhead for the distribution of work/iterations in the parallel regions as the threads/cores increase.
2.) As we add more threads/cores, we may potentially have to wait longer for the implicit synchronization points at the end of parallel constructs because we have to wait for more threads to complete their work before any thread moves on.
3.) Adding more threads/cores means more forks and joins.
4.) Executing reductions takes a little longer since the reduction tree has more levels.
5.) A lot of the code is not parallelized, so when we add more processors these serial portions do not get any faster.

Pseudocode

-Get the users input and set the thread count, chunk size and n.

-Create the randomized vectors (B_copy and B) and matrices (A_copy and A)…we need 2 of each so we have copies to calculate the l2 norm later.

```
for i = 0 to n … SERIAL
```

Partial Pivoting Start/End

```
largest =  largest value in column i for every row greater than i of the A matrix…switch this row (A[largest]) with the row A[i]…also, switch B[largest] with B[i]
```

Forward Elimination

```
for r = i + 1 to n … PARALLEL

        multiplier = A[r][i] / A[i][i]

        for  k = i + 1 to n + 1 … SERIAL

                A[r][k] = A[r][k] – (A[i][k] * multiplier);

        endFor(k)

        B[r] = B[r] – (B[i] * multiplier);
```

Forward Elimination End

```
endFor(r)

endFor(i)
```

-Create zero initialized x vector to store the x values we will soon calculate

Back Substitution

```
for i = n -1 to 0 … SERIAL

        x[i] = B[i]/A[i][i]

        for j = i-1 to 0 … PARALLEL

                A[j][i] = A[j][i] * x[i];

                B[j] = B[j] - A[j][i];
```

Back Substitution End

```
endFor(j)
```

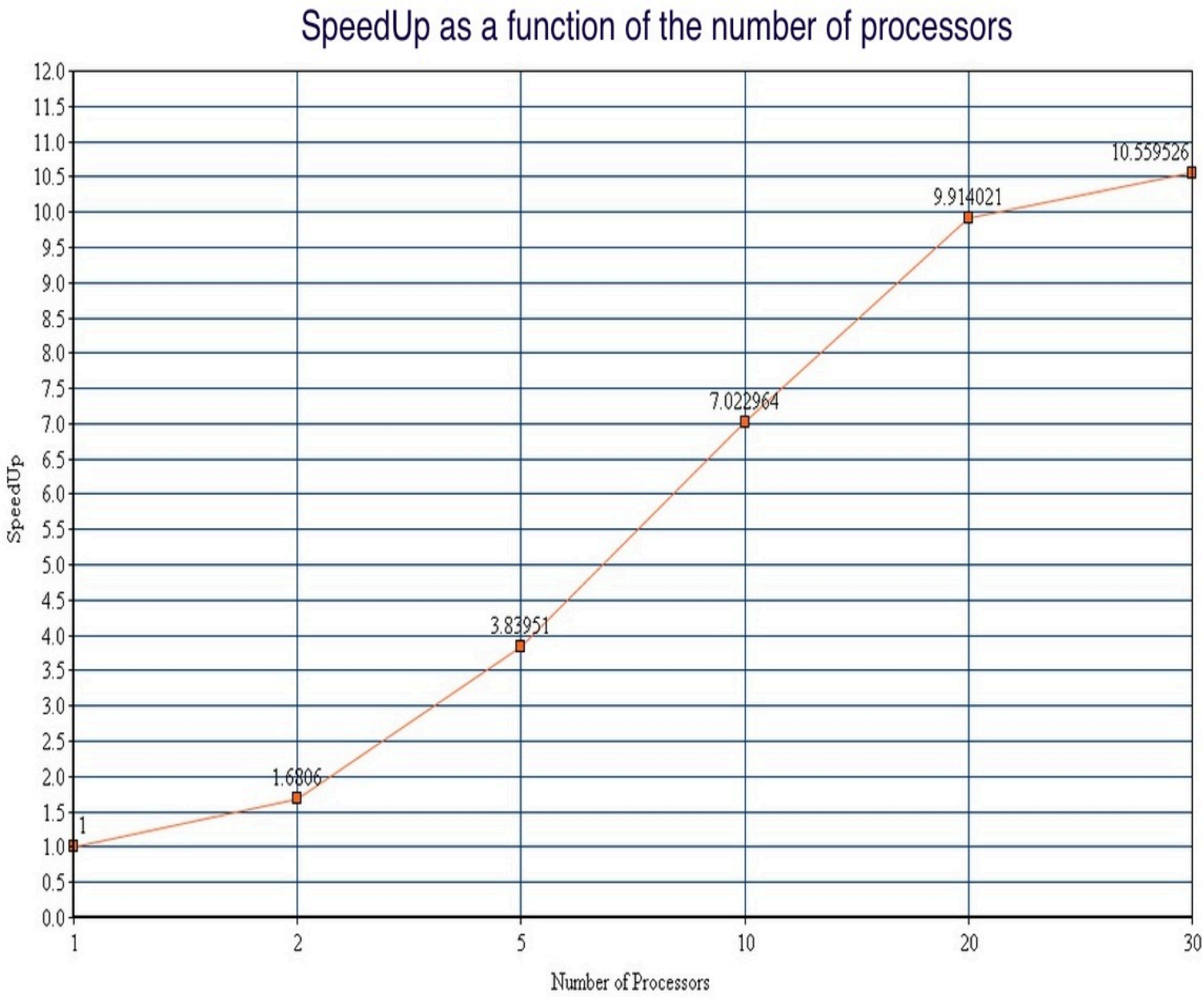
Yaseen Aniss

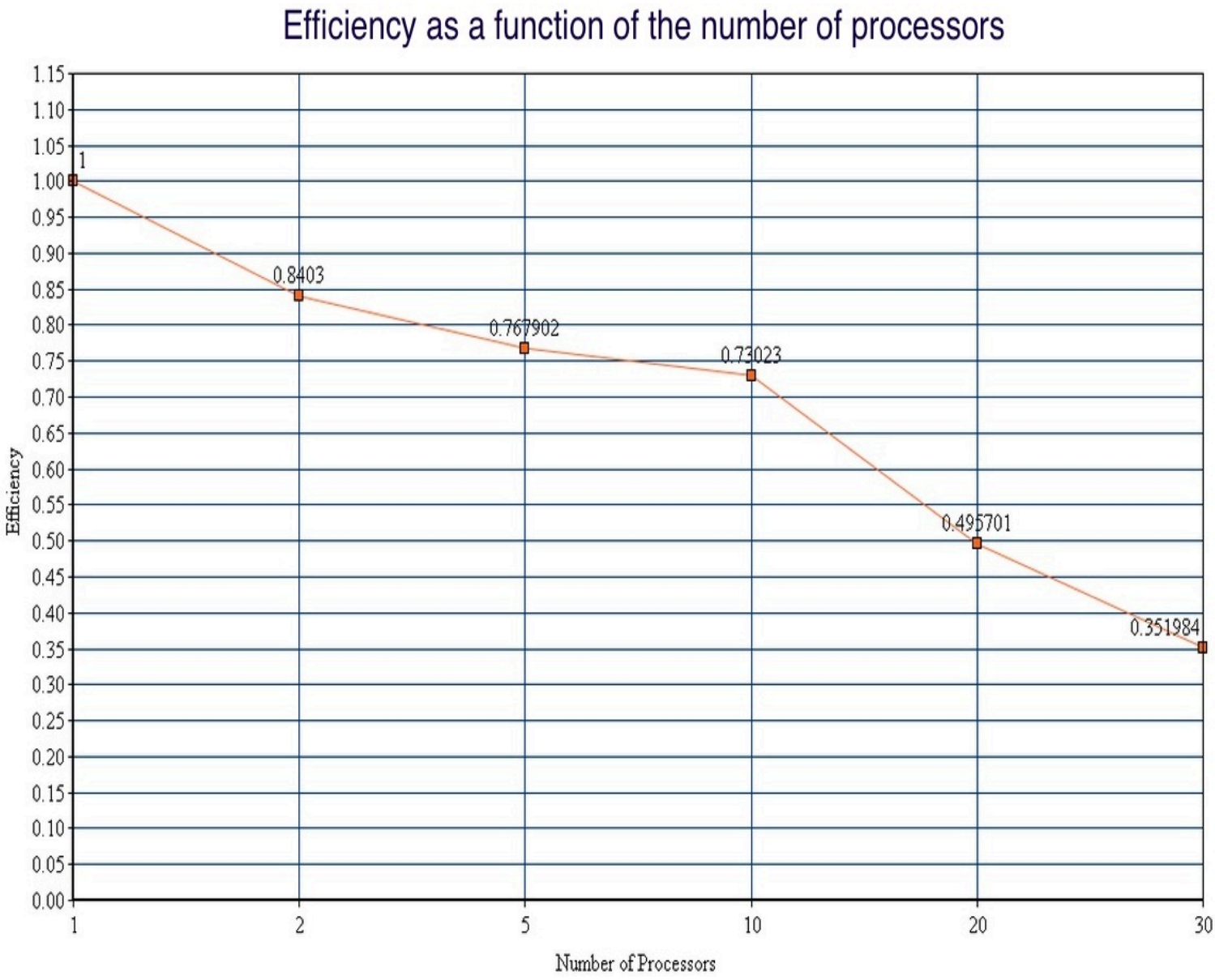
Timing Results and L2-Norm Table

Number of Processors (Times are in seconds) n = 8000					
1	2	5	10	20	30
Time: >2487.500564< L2N: 0.0023266151	Time: 1482.24671 L2N: 0.0021582933	Time: 661.053454 L2N: 0.001260133	Time: 492.949761 L2N: 0.0035262791	Time: 477.886977 L2N: 0.0044774944	Time: > 235.569347 < L2N: 0.012925863
Time: 2496.628454 L2N: 0.0047228771	Time: 1480.628192 L2N: 0.0018320505	Time: 648.375513 L2N: 0.0010149966	Time: 445.772607 L2N: 0.0023262191	Time: 341.415426 L2N: 0.0015791855	Time: 236.120096 L2N: 0.0055294395
Time: 2488.230062 L2N: 0.0011697389	Time: 1486.960814 L2N: 0.0073891378	Time: 647.912637 L2N: 0.0030335835	Time: 484.687154 L2N: 0.010362484	Time: 396.227921 L2N: 0.001158977	Time: 431.884212 L2N: 0.0028378147
Time: 2496.750714 L2N: 0.01856927	Time: 1480.571608 L2N: 0.0089454246	Time: 661.091399 L2N: 0.00217258	Time: > 340.646341 < L2N: 0.0013496729	Time: > 250.90734 < L2N: 0.00097799192	Time: 313.485555 L2N: 0.0014896091
Time: 2488.263981 L2N: 0.0015968391	Time: >1480.126884 < L2N: 0.0015696568	Time: > 647.869312 < L2N: 0.00092725067	Time: 425.400839 L2N: 0.0012685486	Time: 443.498931 L2N: 0.0026668764	Time: 294.208638 L2N: 0.0012689033

SpeedUp and Efficiency

Processors	1	2	5	10	20	30
SpeedUp	1	1.680599542	3.839509787	7. 02296442	9.914020706	10. 55952566
Efficiency	1	.8402997712	.7679019574	.7302296442	.4957010353	.3519841887


SpeedUp as a function of the number of processors
SpeedUp
Number of Processors
12.0
11.5
11.0
10.5
10.0
9.5
9.0
8.5
8.0
7.5
7.0
6.5
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
1
2
5
10
20
30
1
1.6806
3.83951
7.022964
9.914021
10.559526


Efficiency as a function of the number of processors
Efficiency
Number of Processors
1.15
1.10
1.05
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
1
2
5
10
20
30
1
0.8403
0.767902
0.73023
0.495701
0.351984