cb4904, Caraline Bruzinski

HPC
Assignment 3

April 2, 2023

1. Open-MP warm-up.

1a

How long would each thread spend to execute the parallel region? How much of that time would be spent waiting for the other thread?

For this question, both loops are incremented from (1...N-1). There will be a total of $N-1$ iterations to assign to the threads. If N is odd, we compute $N-1$ iterations, and the resulting number of iterations will be evenly divided by the two threads. Both threads are assigned the same number iterations $(N-1)/2$, and both threads will take $i*(N-1)/2$ time to execute each loop (and there are two loops), so each thread takes $i*(N-1)$ time to perform its iterations.

However, if N is even, we still compute $N-1$ iterations, but since we are using two threads the number of iterations will not be divided evenly between the threads. One thread will be assigned one additional iteration on each loop compared to the other thread. Thread 1 will be assigned $N/2$ iterations, and there are two loops, so it will take time Ni to complete its assigned work. Thread 2 will be assigned $(N-2)/2$ iterations, and there are two loops, so it will take time $(N-2)i$ to do its work.

Thread 2 is assigned less iterations, and it finishes first. Thread 2 waits i time at the end of the first loop and i time at end of the second loop. So, thread 2 waits up to $2i$ time for the first thread to complete its overall two additional iterations.

Since we did not specify a value of "chunk size", iterations are divided into sequential iteration chunks that are approximately equal in size (if N is even, thread two will be assigned one less iteration), and at most one chunk is distributed to each thread. When I implement this code, using $N=10$, thread one is assigned iterations $1,2,3,4,5$, and thread 2 is assigned iterations $6,7,8,9$. Thread 1 takes (num loops * num iterations * i) = $5*2*i = 10*i$ time for execution, and thread 2 takes (num loops * (num iterations * i + wait)) =$2*(4*i+i) = 10*i$.

1b

How would the execution time of each thread change if we used 'schedule(static,1)' for both loops?

This schedule works slightly differently than the previous generic schedule. The '(static,1)' schedule does not assign sequential numbers of iterations to a single thread at a time. Using 'schedule(static,1)' changes the thread assignment with every iteration, because we have set the "chunk size" to 1. When I implement this with $N=10$, thread 1 is assigned iterations $1,3,5,7,9$, while the other thread is assigned iterations $2,4,6,8$. The total number of iterations performed by each thread remains the same. When N is even, the first thread will be assigned an additional iteration compared to the other thread. As in the previous example, the first thread waits a total of $2i$ time for the other thread to complete its additional iteration over both loops. Interestingly, when I implemented this I saw marginally worse performance of the thread execution when compared to the static schedule with default chunk size. The compiler may be better able to optimize work and loop execution when the threads are assigned sequential iterations within a chunk.

1c
Would it improve if we used 'schedule(dynamic,1)' instead?

With using the 'schedule(dynamic,1)' directive, each thread executes a chunk (in our case, "chunk size" = 1), and when it finishes that iteration, it requests another chunk of size 1 (i.e. it requests one more iteration). The same amount of work is to be divided between the two threads. However, with this method there is more overhead as the threads must dynamically request iteration assignment once work is finished. The iterations are not assigned beforehand, as in the static schedules. This additional overhead takes more time for each thread to complete the overall work, and the time taken by both threads increases.

When I implemented this with $N = 10$, thread 1 did iterations $1,3,4,5,6,7,8,9$, while thread 2 did iteration 2 in the first loop. In the second loop, thread 1 did all iterations, and thread 2 did no work. This result was a little unexpected, I was expecting that thread 2 would have been more useful in requesting iterations, but this is not the case. In my case, thread 2 waited $8i$ in the first loop and $9i$ in the second loop.

1d
Is there an OpenMP directive that allows us to eliminate the waiting time and how much would the threads take when using this clause?
We can apply the *nowait* directive to avoid waiting for the other thread at the barriers. With this directive, each thread can continue on with its life as soon as it completes the loop execution. If we apply it to the static scheduling case, then at the end of the first loop thread 2 will not wait for thread 1 to complete its additional iteration, and thread 2 will begin the second loop immediately as soon as it completes the first loop. Likewise, at the end of the second loop thread 2 will not wait for thread 1. Overall, this will save $2i$ time for thread 2.

2. Parallel Scan in OpenMP.

I implemented this code and ran it on crackle3 on CIMS:

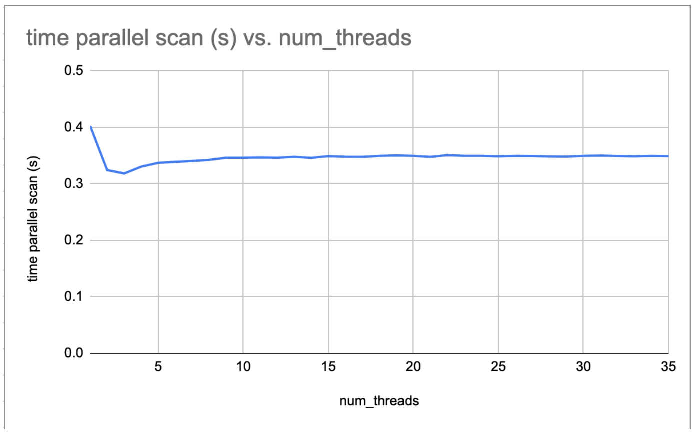
Server: crackle3.cims.nyu.edu
Architecture: x86_64
CPU: Two Intel Xeon E5630 (2.53 GHz) (16 cores)
Thread(s) per core: 2
Memory: 64GB
L1 cache: 32K
L2 cache: 256K
L3 cache: 12288K

With this architecture, my initial assumption is that I should be able to fully utilize (16 cores) * (2 threads per core) = 32 threads. I expect to see a performance increase as I increase the number of threads for this task, and then I expect after 32 threads are applied my performance should taper off and perhaps even get worse. Results for timings with various thread numbers applied are in the table and chart below:

N_THREADS, chunk size, time parallel scan (s)

```
1   100000000   0.40252
2   50000000    0.324276
3   33333334    0.318473
```

```
4   25000000    0.330532
5   20000000    0.337241
6   16666667    0.338966
7   14285715    0.340416
8   12500000    0.342424
9   11111112    0.346277
10  10000000    0.346277
11  9090910     0.346871
12  8333334     0.346442
13  7692308     0.347905
14  7142858     0.346152
15  6666667     0.349143
16  6250000     0.34798
17  5882353     0.347724
18  5555556     0.349485
19  5263158     0.350477
20  5000000     0.349567
21  4761905     0.347897
22  4545455     0.350926
23  4347827     0.349642
24  4166667     0.349487
25  4000000     0.348796
26  3846154     0.349489
27  3703704     0.349389
28  3571429     0.348659
29  3448276     0.34845
30  3333334     0.349612
31  3225807     0.350184
32  3125000     0.34935
33  3030304     0.348797
34  2941177     0.349505
35  2777778     0.349188
```



In the plot above, focus on the sharp dip in time taken by the parallel scan once we add multi threading. The performance significantly improves until the number of threads is 3, and then the process begins to take more time as we continue to add threads, and eventually time taken reaches a plateau at around 10 threads. This was really unexpected for me, I thought I would be able to harness the capability of all 32 threads, but in reality the best performance was with three threads.

Ultimately, this task is still memory bound as all threads compete for the same memory. This exercise demonstrates the importance of tuning the number of threads for a given process because we may not easily get the speedup we expect to see.

3. OpenMP version of 2D Jacobi/Gauss-Seidel smoothing.

In this section, I implemented the smoothing function using both Jacobi and GS iterative methods on the same crackle3 server described earlier. My code is in the github repository, and it compiles and runs with or without openMP.

Timings for different numbers of threads
We want to explore the effect of multithreading on the 2D problem. As implemented correctly, changing the number of threads does not change the results, nor the number of iterations to convergence. This is demonstrated in the figure below. I tested with thread numbers in [1-24] for both methods. I experimented using a smaller size matrix, $N = 100$, as these methods take a long time to converge and we only care about any time difference as we modify the number of threads.

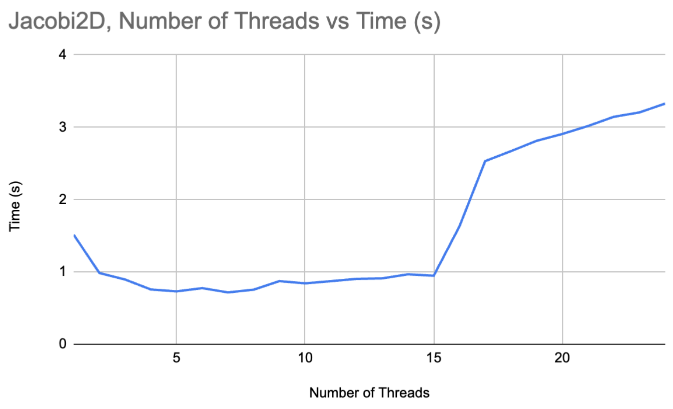
Jacobi 2D runtime
Threads, Iterations, Time (s)

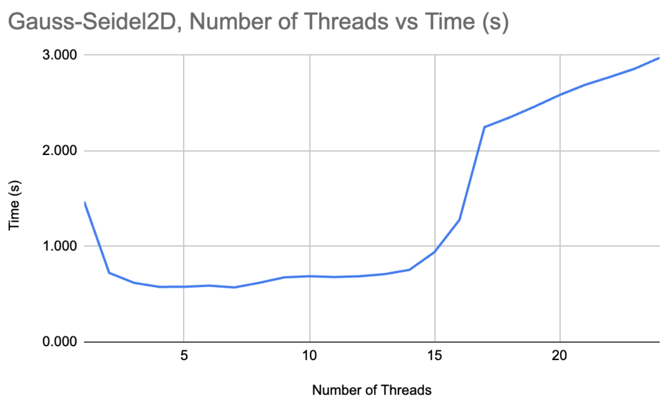
```
1       23381   1.515
2       23381   0.985
3       23381   0.895
4       23381   0.758
5       23381   0.731
6       23381   0.776
7       23381   0.717
8       23381   0.755
9       23381   0.873
10      23381   0.842
11      23381   0.872
12      23381   0.904
13      23381   0.911
14      23381   0.967
15      23381   0.948
16      23381   1.630
17      23381   2.532
18      23381   2.670
19      23381   2.813
20      23381   2.908
21      23381   3.018
22      23381   3.145
23      23381   3.206
24      23381   3.328
```

Gauss-Seidel 2D runtime
Threads, Iterations, Time (s)

```
1     14812        1.469
2     14812        0.724
3     14812        0.619
4     14812        0.577
5     14812        0.578
6     14812        0.590
7     14812        0.571
8     14812        0.620
9     14812        0.677
10    14812        0.688
11    14812        0.680
12    14812        0.688
13    14812        0.710
14    14812        0.755
15    14812        0.940
16    14812        1.278
17    14812        2.248
18    14812        2.349
19    14812        2.463
20    14812        2.584
21    14812        2.689
22    14812        2.772
23    14812        2.859
24    14812        2.974
```

Thread Performance Comparison





Focusing on the charts above, both algorithms follow a similar trend as the number of threads increases. Initially, it is trivial, that the runtime of using one thread is the same as the runtime without openMP (because it is single threaded). Then, as more threads are added there is a sharp decrease in the amount of time the process takes to complete, but this performance gain soon tapers off. Based on my results, I achieve the best performance when the number of threads is set to 7 in both methods.

Conversely, when too many threads are in use (around 15 threads) the performance degrades rapidly for both methods. The total time taken by the process increases so much that it surpasses the initial, single threaded runtime. This may be due to too many threads competing for CPU and memory resources, and/or too many resources spent attempting to manage the threads.

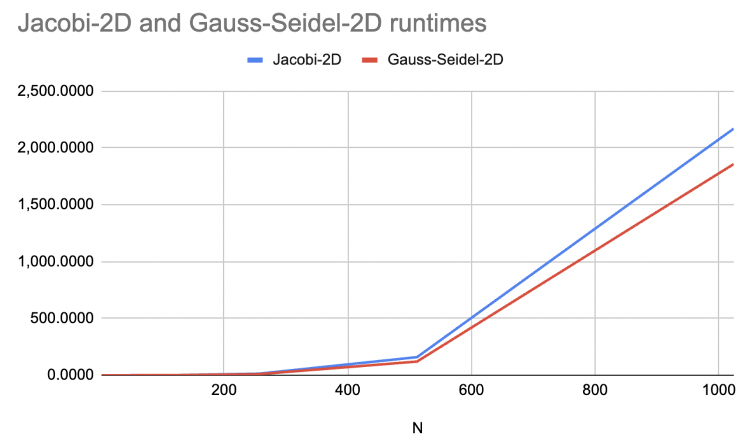
I was not entirely surprised that both methods perform best at the same number of threads (7) considering the implementation of each method is similar in the 2D case. However, as in problem 2 I would have liked to see the performance continue to improve as we increase the number of threads to the number of available threads across the 16 cores. As in problem 2, we really see best performance with a much lower thread number.

Timings for different sizes of N

Considering that the best performance for both algorithms was achieved when the number of threads is set to 7, I used 7 threads in the next part to experiment with different sizes of *N* and compare the total runtime of each method.

Multithreaded time to convergence, varying size N

N	Jacobi-2D (s)	Gauss-Seidel-2D (s)
2	0.0003	0.0004
4	0.0020	0.0009
8	0.0044	0.0043
16	0.0130	0.0127
32	0.0499	0.0423
64	0.2310	0.1914
128	1.5100	1.0970
256	13.1340	9.0217
512	158.8660	120.1670
1024	2,169.7810	1,858.1980



As we have seen from HW1, Gauss Seidel will converge in less iterations, and is faster than, Jacobi. Gauss-Seidel simply takes less iterations to converge than Jacobi. We can say the same is true for the 2D problem. When *N* is small both methods converge quickly, and the runtimes are similar. Based on the results for small N, Gauss-Seidel is marginally faster. However, the difference in performance is amplified with larger N. Eventually, as in the $N = 1024$ case, Gauss-Seidel converges 14% faster than Jacobi.

I attempted to run both methods for $N = 2048$, but after letting Gauss-Seidel run for three hours, the residual was still about 1/4 of the initial residual, so I believe it would have taken a few more hours for GS-2D to converge. These methods take an extremely long amount of time as we increase N, even with multi-threading applied and using our best performing number of threads (7).