Hardware Aware Scientific Computing

Purusharth Saxena

July 2021

1 Task 1

1.1 For large n and k

Any analysis on n obviously depends on k. At a distance k away from boundary, the floating point operations would be

$$4 * (k+1)^2 (n-2k)^2$$

Where $4 * (k+1)^2$ is the flop for each box of size k. Similarly, the complexity can be written down as: $O(4k^2 - n^2) \sim O(k^2 n^2)$

Hence, for large n, the complexity and consequently, the FLOPs will increase quadratically.

An increase in k could provide good caching opportunity, as a lot of elements needed for (i+1,j) would be in cache from the previous iterations of (i,j). This would also be valid for (i,j+1) or (i+1,j+1).

Reuse of in-cache data between threads:

Furthermore, the local mean operation can be tweaked slightly to reduce data transfer and FLOPs. If α is stored in a temp variable, then the mean of (i+1,j) can be calculated by $\frac{4k^2 u_{ij} - \alpha + \beta}{4k^2}$ where $\alpha \& \beta$ are sums of respective columns. This makes sense for large k for points at a respectable distance from the boundary.

1.2 Spatial Blocking

A suitable blocking could be as shown in figure 1 .

Theoretically, the efficiency of blocking would also depend on k. For a small block and large stencil, we'd not observe good result the elements would constantly have to be accessed outside the block. If the blocking size is optimal, then at-least $2(k+1)$ elements would be recently accessed for the last element in the block. For the next block, there would be some elements that have been accessed recently.

However, since k is dynamic, there might not be an impressive speedup after all (or any speedup for some combinations of k and b).

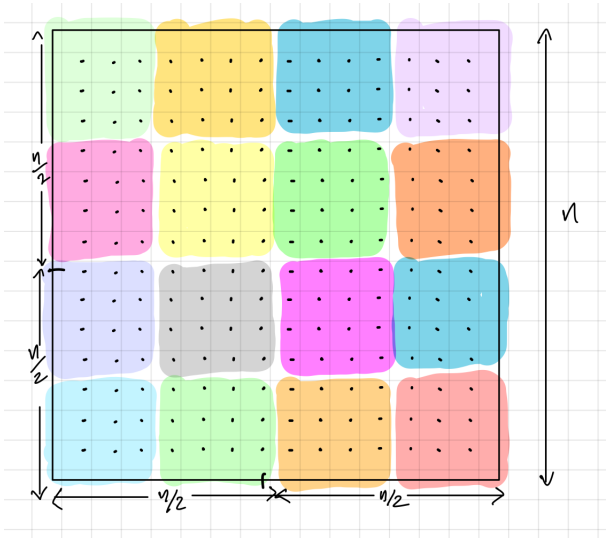

Figure 1: Spatial blocking

1.3 psuedocode

```
If block_size % n == 0
    For i, j = 0 to n
        For blockRow = i < i+blocksize, blockCol = j < blocksize
            r, c <- row, col # for the k_stencil
            For k_stencil in blockRow, blockCol
                # calculate sum
                blockRow++, blockCol++
            Endfor
            i++, j++
        Endfor
        i+= blocksize, j+=blocksize
    Endfor
EndIf
```

1.4 Intensity

The matrix M consists of b^2 blocks. For each block, n_{ij} we load $(2k+1)^2$ elements, however, each element is acessed atmost $2k+1$ in the same row and can be assumed to have come from cache/register instead of main memory. Each loop for computing an element i, j in b_{ij} employs $(2k+1)^2$ FLOPs, for a total of b times, Hence, $I = \frac{b^2(2k+1)}{(2k+1)^2} = \frac{b}{2k+1}$

1.5 SIMD vectorization

Since the primary operation is just a single instruction, SIMD vectorization can of-course be added. SIMD vectorization can be added in the innermost loop, which loops over individual points. For large enough b and k, SIMD operation can be performed to add elements within the same row as shown in the diagram below. (Say k is 7, then 8 elements are present in each row. These 8 elements can be loaded up in two 4dvec and added before moving to the next row).

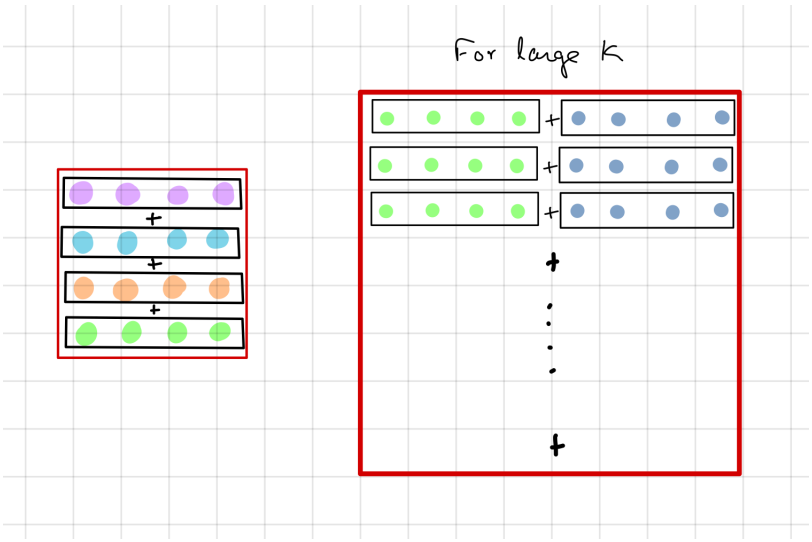

Figure 2: SIMD vectorization, right one is a better if k is expected to be large

2 Task 2

2.1 Grid Points

A point is close to grid if it the distance is less than k from n. The number of points in the stencil would be less than $(2k+1)^2$.

Since the number of neighbours inside the k stencil are constant for most parts near the boundary, value of the previous point can be used for calculation.

Consider a point at the boundary: $\mu_([0][0]) = \frac{\Sigma u_{r,s}}{(k+1)^2}$, then $\mu[1][0]$ can be calculated just by $\frac{\mu_[0][0]*(k+1)^2+\beta}{(k+1)(k+2)}$ where β consists of k+1 points.

Same thing can be done for boundary values that are k+1 distance away from the corner. $\mu_{i+1,j} = mu_{i,j} - \alpha + \beta$

Since the number of points "near" the boundary will always be $n^2-(n-2k)^2$, boundary values can be calculated before starting the main loop for iteration. Things become more simpler if boundary values remain constant.

An alternative approach would be to calculate the number of points at the boundary compute the points in the middle outside the loop.

2.2 Performance comparision

```
Flags: -O3 -funroll-loops -ftree-vectorize -march=native -fopt-info-vec
```

2.2.1 System Arc

```
CPU
    Name:        Intel(R) Processor code named Kabylake ULX
    Frequency:   1.8 GHz
    Logical CPU Count:   8

Clock Rate
            CPU max MHz:                          3400.0000
            CPU min MHz:                          400.0000

Cache
    L1d (Data): 128K
    L1i (Instruction)   : 128 KiB
    L2 cache (Unified) :   1 MiB
    L3 cache (Unified) :   6 MiB

Multithreading
    Thread(s) per core: 2

Vectorization
            SSE: 1,
            SSE2: 1,
            SSE3: 1,
            SSE4.1: 1,
            SSE4.2: 1,
            SSE4a: 0,
            SSE5: 0,
            AVX: 1,
```

Varying b and k gives quite interesting result. Some of them are presented below. The value of n also has some significance.

If $n = 500$: Blocked and Vanilla versions have better performance than SIMD implementations. If the size of block $> (2k+1)$, then blocked version has the best performance (assuming that $2k+1$ is not large < 15). The constraint with k takes away the glory of blocking. However, if blocking size $<$ k, then SIMD implementations perform better than blocked version for n=[400,500]. If $k < b$ but k itself is large, then we have an obvious speedup with SIMD implementations, however, if k is small enough then blocked version outperforms the rest. One such comparison is shown in figure 5

However, if n > 500, the SIMD operations, unsurprisingly, outperforms the blocked version, even if blocking size is smaller than k One such result is shown in figure 3

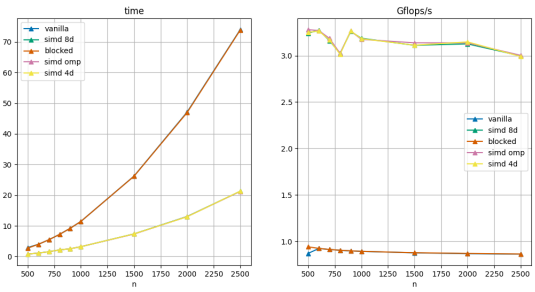

Figure 3: n = [500, ..., 2500], k = 50 (2k+1) = 101, b = 50

Since there were no Vec8d registers on the laptop used for benchmarking, the VCL library emulated it using two 4d registers and hence the performance is almost the same as with Vec4d. In some cases, the overhead of emulation actually makes it worse.

As discussed earlier, if the value of block size is smaller then stencil, then the performance would not be near optimal. Two such cases were analyzed by vtune and are shown in table 1 and Figure 4 (Note that the value of n is different with vtune and benchmark).

k = 25, b = 100	**memory bound**	**Processor Usage**	**Vectorization**
Blocked	46.9%	16.3%	0%
SIMD 4d	0.1%	89.3%	77.7%
SIMD OMP	9.1%	58.6%	83.2%
k = 50, b = 20	**memory bound**	**Processor Usage**	**Vectorization**
Blocked	47%	12.4%	0%
SIMD 4d	6.3%	59.2%	93.0%
SIMD OMP	11.5%	58.6%	63.1%

Table 1: For **k = 25, and b = 100** (top) and **k = 50, and b = 20** (bottom); n = 5000 in both cases. In the topmost table, since B > $(2k+1)^2$ We observe proper usage blocking and less memory boundedness, with the VCL library providing exceptional cache utilization. The blocked version has high memory bandwidth, but all of it is concentrated on L1 cache.
In the bottom-most table, we see clear difference in memory bandwidth - especially with the VCL library. Even though the blocked version has the same memory boundedness as shown in topmost table, around 4.8% of it is L2 bound

.

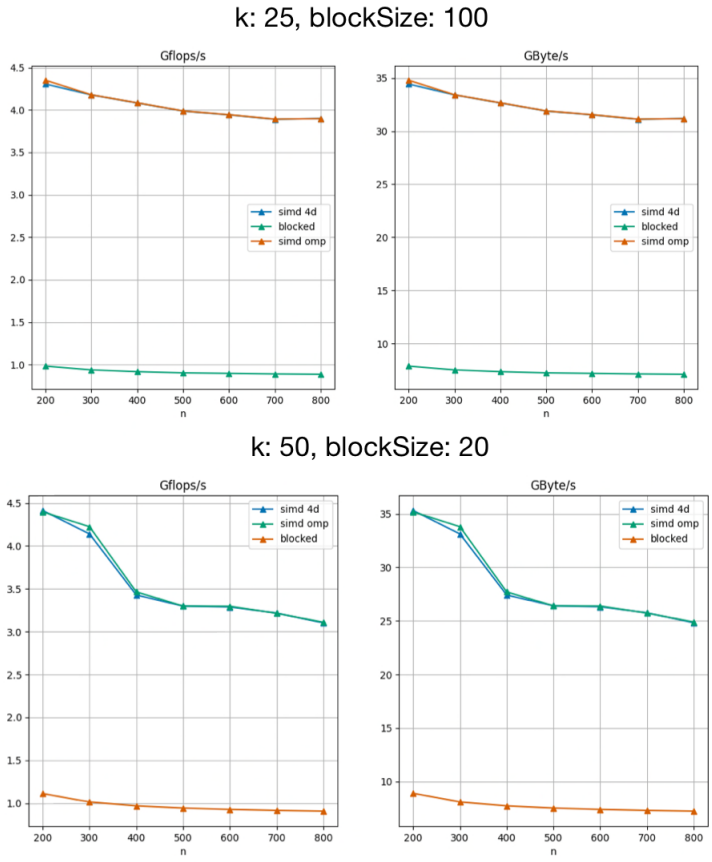

Figure 4: Gflop/s and Gbytes/s for n = [200,..,800]. The combination of k and b are the same as used in the VTune analysis shown in Table 1

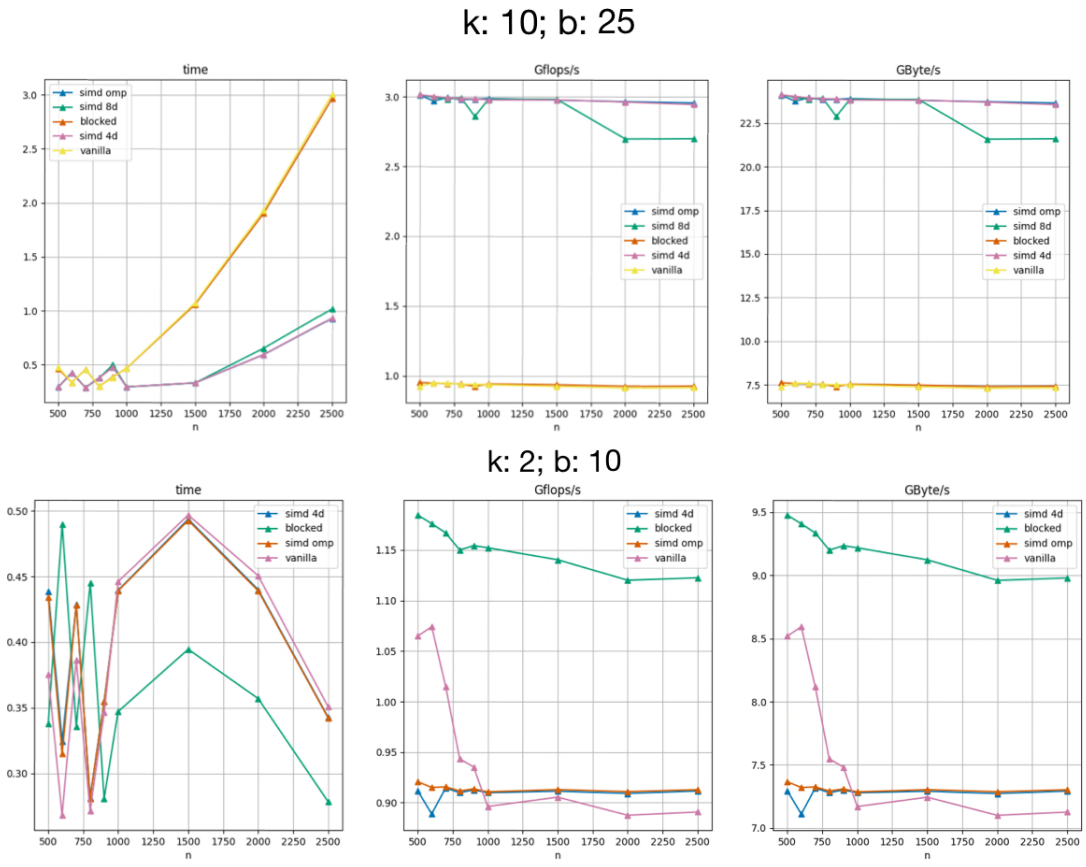

Figure 5: $k < b$ in both cases, but the size of k differs. Blocked version outperforms the rest when k is small enough.

3 Task 3

3.1 Jacobi

advantage disadvantage
Since Jacobi is naturally parallel, any of the aforementioned methods can be used to parallize it. The only condition with Jacobi parallelism is that all threads/rank should wait until all threads/rank are done computing the i^{th} iteration before starting the $i + 1$ iteration.

It's worth noting that that k will be an important factor, especially in distributed memory paradigm.

3.1.1 C++

The data can be equally divided amongst all available threads in the system. The threads can start execution in any order. However, there needs to be a barrier at the end of the loop (before the next iteration can start). The parallization can also be done in terms of iterations (wavefront). Advantage: Straightforward implementation. Disadvantage: Choosing a wrong barrier can hamper performance. Wavefront iterations for iterations will be complicated.

3.1.2 OpenMP

It is perhaps the easiest method to parallelize Jacobi. OpenMP will also use the same paradigm as C++ threads, however, the distribution of threads and load balancing will be taken care of by OpenMP itself. Essentially, openMP parallization is done by just adding #pragma omp for before the first loop (slower index). The barrier in OpenMP for is implicit. Advantage: Easiest implementation Disadvantage: Difficult to parallize over iterations (wavefronts).

3.1.3 TBB

A naive TBB implementation would be the same as OpenMP using $parallel_for$, However, TBB can also be used to parallize Jacobi Wavefront (with respect to iterations) using continue nodes. FOr this, the matrix can again be divided into blocks. A dependency graph can be created using the "continue_node" in TBB flow graph and delegate each block to a node. The "make_edge" function will specify the dependency between two nodes (which would be dependent on k - see figure below). Finally, calling "wait_for_all" waits until all computations are completed. The dependence graph should be acyclic. Advantage: TBB takes care of the dependency so wavefront parallization would be easier. Disadvatage: Can become compilcated quickly.

Note: the Jacobi Wavefront in this case will not depend on 'k' slower indices above and below that point.

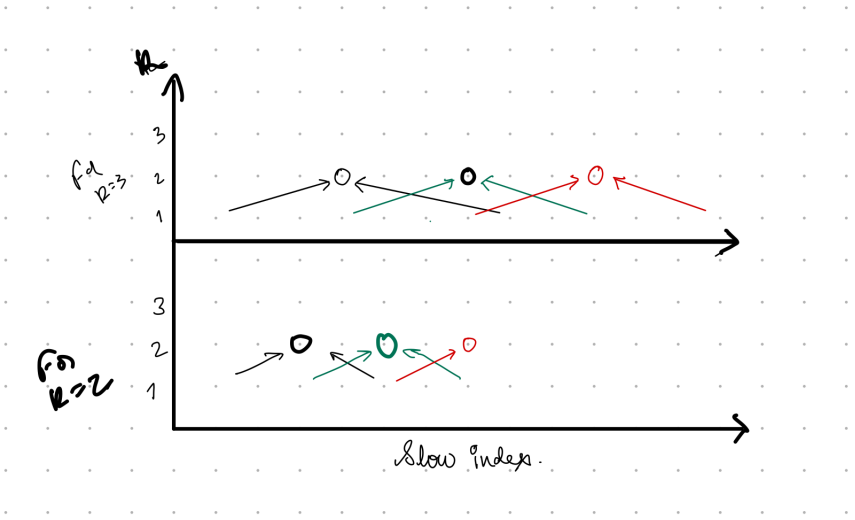

Figure 6: The wavefront (iteration) depends on k rows to the right and left of the give row

3.1.4 MPI

MPI is a little tricky when compared to others since it requires data transfer and some ghost cells. The squared data decomposition will have minimum data transfer. And unlike the traditional finite difference jacobi method, the data transfer and ghost cells will depend on k (as shown in figure 7).

Advantage: Optimal choice for large very large matrix computation. Additionally, if there's large N and small k, then MPI+X (C++/OpenMP paradigm discussed above on each rank) can be used to make computation even faster. Disadvantage: MPI will makes more sense for large N and smaller k, otherwise the application will be communication bound.

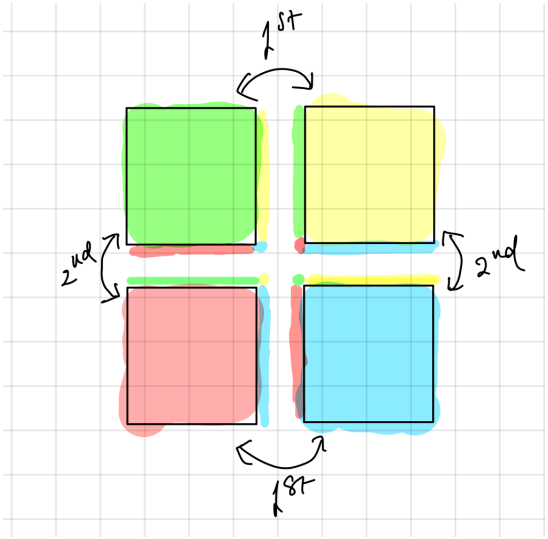

Figure 7: Jacobi Data Decomposition for MPI. The transfer from each side would be of size k*b where b is the size of the decomposition and k is the stencil size

3.2 Symmetric Gauss Seidel

3.2.1 Data Dependency

Unlike Jacobi or Gauss Seidel, Symmetric Gauss Seidel has a *very* complicated data dependency as shown in the figure below.

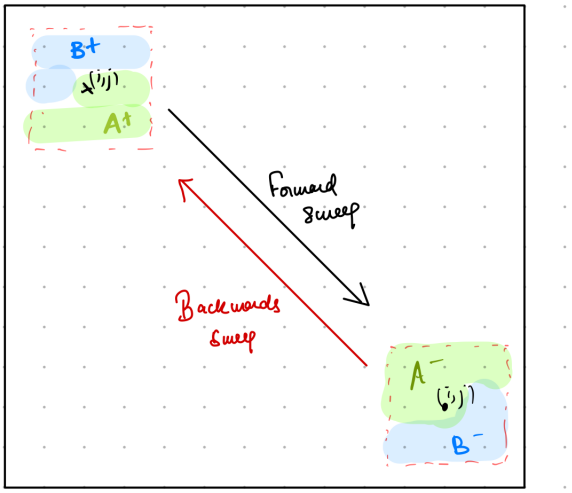

Figure 8: SGS dependency for forward and backwards iteration. B+ and B- dependes on the previous iteration's value.

the data dependency - especially the diagonal one, leads to some very complicated parallization scheme, and all programming paradigm might not be a suitable for this algorithm. Owing to the dependency of the backwards sweep, it is not possible to parallalize with respect to iteration.

3.2.2 C++

C++ threads can be used for parallization with extensive use of barriers. The data is divided vertically as shown in figure 9.

Similar to this, there can also be a p-width wavefront approach as shown in Figure 11.

The scheme discussed in openmp can also be done with threads but it will be much more complicated. Advantage: Possible to parallize SGS with a simple scheme. Disadvantage: Lots of barriers.

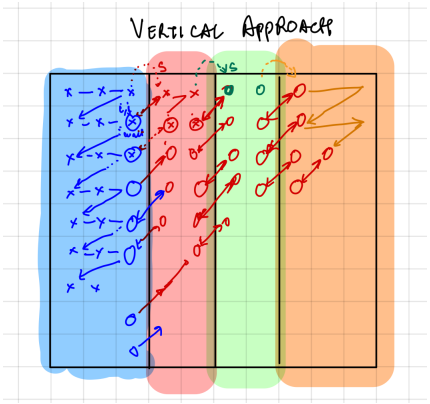

Figure 9: The execution starts at (0,0) while the other threads immediately go to the barrier. Once the first thread reaches (4,0) the second thread starts execution while the third thread waits and so on. while the data is being distributed row wise amongst threads. The first thread will wait at, say, [2][4] for the data at [1][3] but that would have already been computed since [1][3] depends on [0][2] (which was computed in the previous row of Thread 1). This is similar to a sort of temporal pipe-lining in vertical direction.

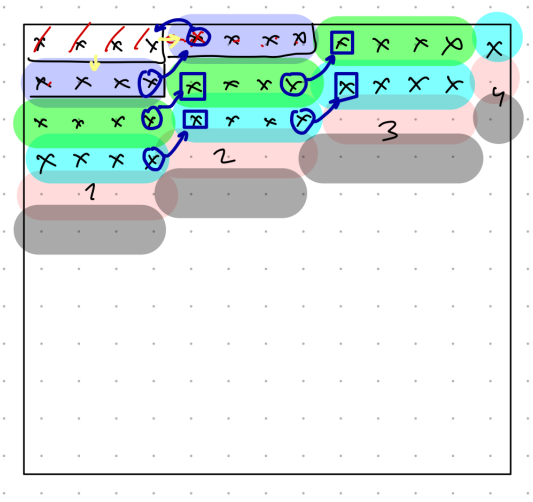

Figure 10: The execution starts with the first block of width p. Once it is done executing, the two surrounding blocks of width p will start execution on different threads. However, the element at [i = 1] [j = p] has to check if k elements have been computed by thread 2 (which probably would have been since it would also be the first few elements for that thread). From a bird's eye view, this also looks similar to temporal pipe-lining in vertical direction.

3.2.3 OpenMP

At first sight, the Symmetric Gauss-Seidel seems to be a fully sequential method. But a more careful analysis, however, reveals a sequence of elements with no interlinked dependency that can be executed in parallel. (I am not sure what this scheme is called, so I refer to it as skewed/off-center wave-front in the text). This approach is illustrated in figure 11.

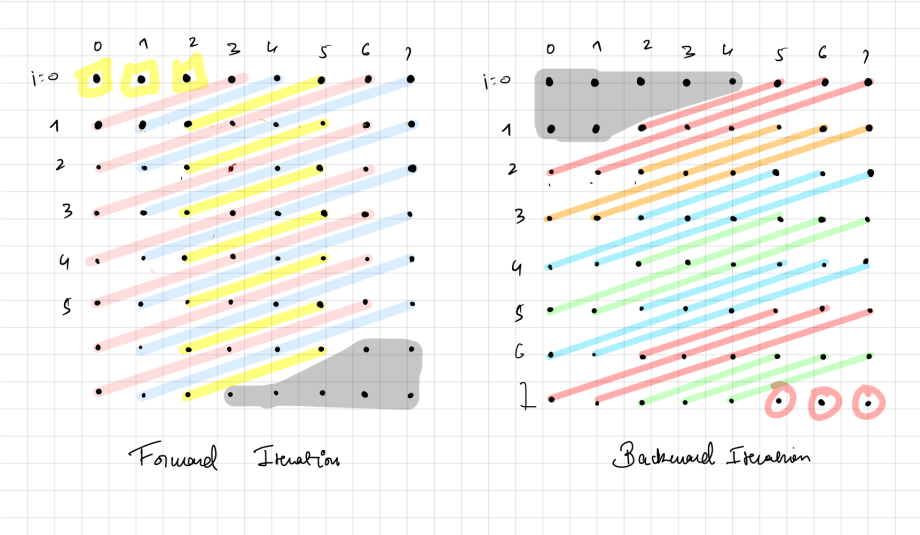

Figure 11: Off-center wavefront for k = 2 with SGS. After the first (k+1) elements, the dependency of element [1][0] and [0][3] is already met, and they do not depend on each other, so they can be executed in parallel. The wave moves forward in a set of (k+1) elements (Compare fig 1 & 2). The loop stops at a few (relative) points away from the bottom corner. The remaining points have to be executed in the exact same manner but in another loop.

Advantage: Easy implementation without too many barriers. Good performance for large N. Disadvantage: Complicated loop structure. Some OMP overhead for smaller values even if n large. The said overhead is repeated during the backwards iteration.

3.2.4 TBB

The Tbb can parallize all the methods discussed above - however, the off-center wave-front will be more complicated in TBB's "parallel_for". As for the vertical and k-block wavefront, a combination of function, continue and join nodes can be used as shown in the figure below for vertical and k-width wavefront algorithm.

Advantage: Theoretically faster computation. Don't have to worry about barries (as with C++)

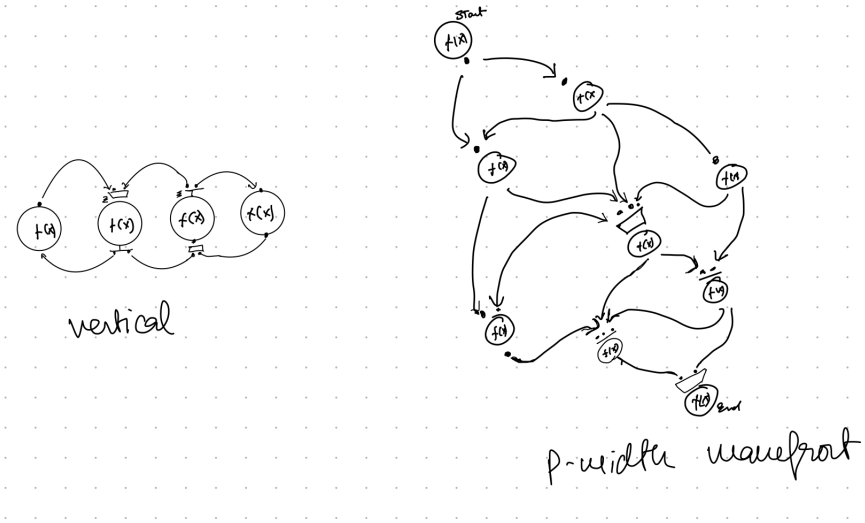

Figure 12: TBB flow graph for vertical and p-width wavefront

3.2.5 MPI

MPI can use the same vertical data distribution as discussed in C++ threads. however, there's a good chance that the program will become communication bound, except in the case where $n >> k$. If not, then MPI does not seems to be a suitable model. And even otherwise, a synchronization before a backward sweep would add extra communication overhead.

4 Task 4

4.0.1 Jacobi

An unsurprising speedup is observed with the jacobi's parallel implementation when compared to sequential one. Benchmark for different values of k is shown in the Figure 13.

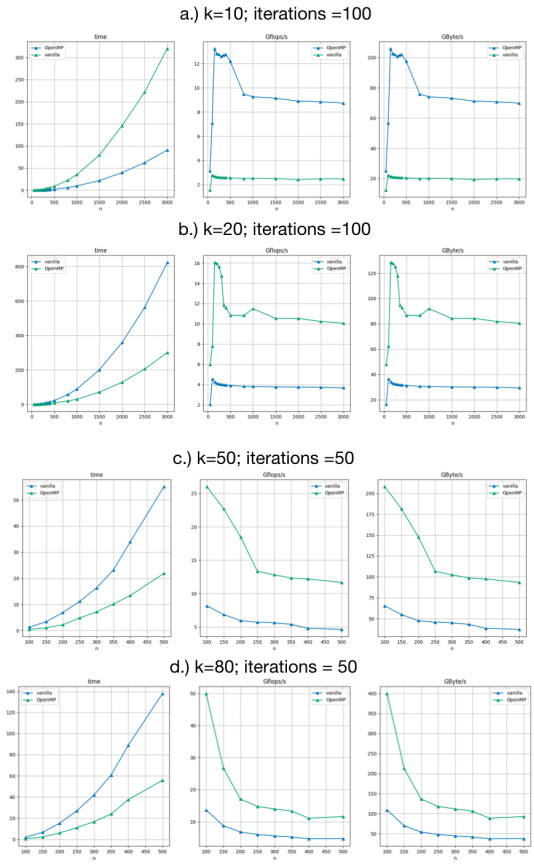

Figure 13: Jacobi Seq vs Parallel performance. a&b have small values of k while c&d have larger values of k. The effects of caches are shown in Gflops/s graph.

The speedup happens because the data in the array is equally divided amongst all the threads. For smaller values of n, the performance of omp version is actually equal to (or less than) the sequential version. This happens because of omp overhead. The capacity of L1 and L2 cache can be observed from a & b

of figure 13

Intel VTune report of the sequential and parallel version are given in table 2

	Sequential	**Parallel**
Time (s)	275	40.757
memory bound	40.9%	8.9%
Procesor usage	22%	58.3%
Vectorization	0%	88.4%

Table 2: Jacobi performance comparision by VTune. n = 500, k = 50, and iterations = 100 in both cases. The parallel version is around 6x faster for this configuration. We see good vectorization with the parallel version provided by omp simd. There is also less memory boundedness in the parallel version

4.0.2 SGS

The SGS implementation also uses OpenMP operations. A parallel implemetation gives anywhere between 3-5x speedup - depending on the configuration. The results are shown in figure 14. It is interesting to observe that for relatively smaller values of k, parallel version is around 4.5x faster for large n, while with larger values of k the parallel version is around 3x faster. This difference can be reasoned with by the number of operations performed per element, and as we can see in table 3, the Parallel SGS implementation did not properly exploit the vectorization.

Moreover, for smaller values of n, the both performance are almost similar. Not only it's the openmp overhead, but also the parallization scheme. We can easily conclude by looking at Figure 10 that for smaller values of n, thread creation time for two elements would be larger than computing it. We also have to remember that in the given scheme, 2(k+1) [k+1 from forward and k+1 from backwards sweep] elements contribute to sequential part in Ahmdal's law.

Another interesting observation would be to consider the inner loop, c = 0, 1, 2. If N is very large and k is small, the distribution of data amongst c=0, c=1, and c=2 would be uneven and not all threads will get to run at the same time. Similarly for large n and large k, if the distribution is uneven - then few of the threads will finish early and will wait for the other threads. This was observed (via system monitor) for some values of k cases. Furthermore, each "drawback" will be repeated twice per iteration.

Vtune analysis of both the algorithm is given in table 3.

VTune

Unfortunately, hardware constraints of the laptop prevented from further benchmarking as the vanilla version took too long and led to overheating.

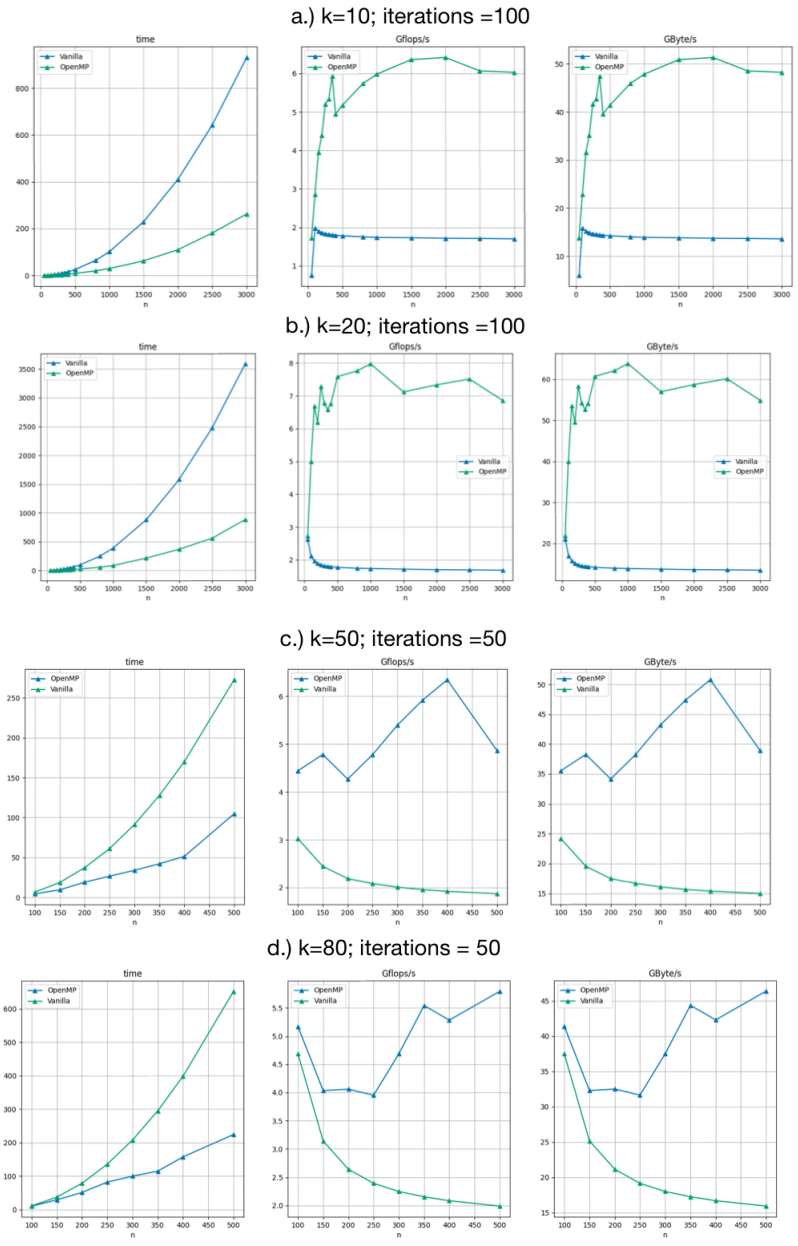

Figure 14: Parallel vs sequential implementation of the Symmetric Gauss seidel.

	Sequential	Parallel
Time (s)	109.8	34.3
memory bound	52.5%	24.1% (L1+L3)
Procesor usage	22.1%	53.3%
Vectorization	0%	20%
Logical Core Utilization	12.4%	94.4%

Table 3: For n=500, k = 50, and 20 iterations. The parallel version is 3x faster than the sequential version. However, it could not exploit vectorization. One reason for that could be the if conditions in the inner loop

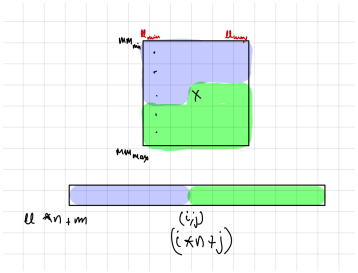

Figure 15: Inner loop can be removed and instead two different for loops with which traverses B+/- and A+/- part respectively. This can improve the packed Floating point operations (since if conditions are removed) by either converting it vec4d loop or adding pragma omp simd

.

Possible Improvements: SIMD 4d vectorization can be implemented instead of omp simd to see if there's any performance improvements.

We can get rid of if conditions and the inner loop (of the k-stencil) that iterates over that the slower index and replace it with two for loops that go over the violet and green part in figure 15. This can be followed by the vectorization of those two for loops separately.