Parallelizing FFT with a Focus on Bandwidth Efficiency

Summary

We have parallelized and optimized one-dimensional (1D) and two-dimensional (2D) versions of the Fast Fourier Transform (FFT) with a focus on improving its memory bandwidth usage using OpenMP. For this project, we implemented parallel versions of 1D Discrete Fourier Transform (DFT), 1D recursive FFT, 1D iterative FFT, and 2D FFT using our iterative 1D FFT; we also implemented several optimizations for these implementations. Finally, as a means of visualizing correctness, we applied our 2D FFT implementation to image compression. At 8 threads, our most performant 1D implementation (iterative) achieved a ~6x speedup, while our 2D implementation achieved a ~7x speedup. At 64 threads, 1D iterative achieved a ~22x speedup while 2D achieved a ~40x speedup. Our iterative implementation was much more performant than our baseline recursive implementation; on a single thread, it was 3.2x faster, and at 64 threads, it was ~260x faster.

Background

What is DFT and FFT?

FFT is an algorithm, with roots in signal processing, for computing the so-called Discrete Fourier transform (DFT) of a sequence of numbers. In the context of signal processing, FFT and DFT are used to decompose a signal into sinusoidal curves with different amplitudes and frequencies. This is known as converting a signal from the time domain to the frequency domain. The naive approach to computing the DFT takes $O(n^2)$ time while FFT computes the result in O(n log (n)) time. Another way of interpreting the DFT is that it evaluates a polynomial whose coefficients are given by the input at n powers of the n^{th} root of 1 (unity). The latter formulation, as shown below, allows us to easily derive the FFT algorithm as well as some of the optimizations we implemented.

Formulating the Fourier transform and the FFT algorithm

As mentioned in the previous section, we will focus on formulating the Fourier transform as a polynomial evaluation problem. We want to evaluate the following polynomial P(y) at specific powers of a principal n^{th} root of unity.

$$P(y) = x_0 y^0 + x_1 y^1 + \ldots + x_{N-1} y^{N-1}$$

By convention, the root we select is $\omega_N = e^{\frac{-2\pi i}{N}}$. DFT can then be formulated as follows

$$DFT([x_0, x_1, \ldots, x_{N-1}]) = [P(\omega_N^0), P(\omega_N^1), \ldots, P(\omega_N^{N-1})]$$

Each element X_k (Fourier coefficient) of the resultant DFT vector can then be computed as follows

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}$$

Note that it's also possible to invert this transformation to go from the point-value representation back to the list of polynomial coefficients in a similar manner.

Therefore, to calculate a Fourier coefficient, we have to access all the elements in the input array. This formulation naturally leads to an $O(n^2)$ algorithm where we compute each Fourier coefficient by iterating through the input vector. Although this is a valid approach, it is not very efficient and doesn't scale to large datasets due to its time complexity.

For the purposes of our project, we only consider the case where N is a power of 2. This greatly simplifies the FFT formulation, but note that it's still possible to compute FFT for non-powers of 2 elements. The FFT algorithm (specifically Cooley-Tukey) utilizes a divide and conquer approach to compute the transform. Observe that the computation of X_k can be split into the sum of 2 terms as follows

$$X_k = \underbrace{\sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2} mk}}_{\text{DFT of even-indexed part of } x_n} + e^{-\frac{2\pi i}{N} k} \underbrace{\sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{2\pi i}{N/2} mk}}_{\text{DFT of odd-indexed part of } x_n} = E_k + e^{-\frac{2\pi i}{N} k} O_k$$

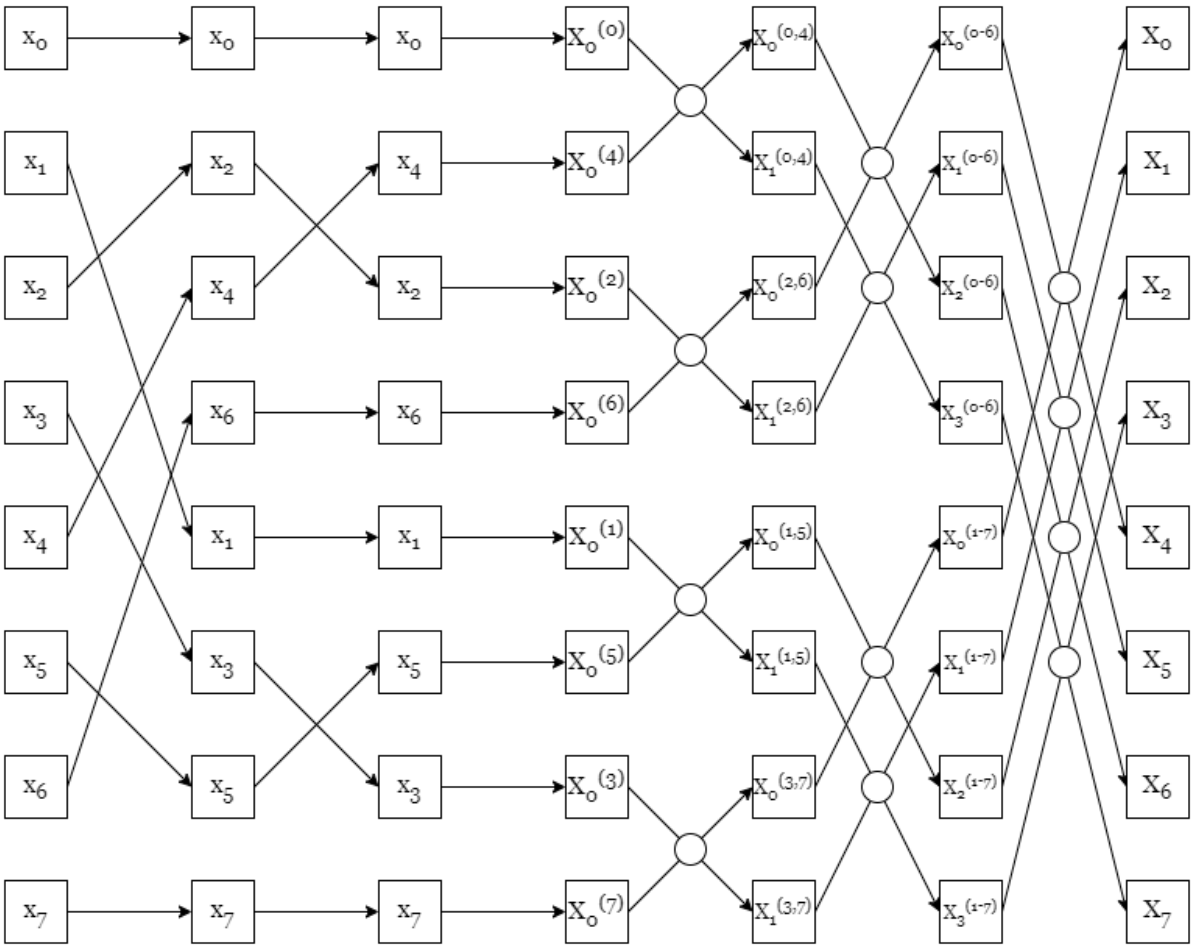
This formula can only be used for $k = 0$ to $\frac{N}{2} - 1$ since we only compute E_k and O_k for those values. But using the fact that $\omega_N^{k+\frac{N}{2}} = -\omega_N^k$ and $\omega_N^{k+N} = \omega_N^k$ we can extend this to indices $\geq \frac{N}{2}$ as well. The final recursive formulation is then

$$\begin{aligned} X_k &= E_k + e^{-\frac{2\pi i}{N} k} O_k \\ X_{k+\frac{N}{2}} &= E_k - e^{-\frac{2\pi i}{N} k} O_k \end{aligned}$$

This formulation directly leads to the Cooley-Tukey algorithm.

1. Split the data into 2 parts. One part with elements from the even indices, and the other with the odd indices.
2. Perform FFT recursively on these 2 parts.
3. Combine the results of the 2 $\frac{N}{2}$ sized FFTs to get the transform of the original input (using the formulation above).

The data flow and dependencies for this algorithm on an 8 element vector are visualized below.



This visualization will help us create an iterative, in-place implementation of the FFT algorithm; we will explain this further in Iterative FFT.

Finally, the 2D FFT algorithm simply performs FFT on each row of the input followed by FFT on each column of the input (or vice versa). We utilize 2D FFT to perform image compression, which is elaborated on in the Image Compression section.

Challenges in parallelizing FFT

The quadratic DFT algorithm is compute bound. It has very good spatial locality since it simply iterates over the input elements repeatedly (once per output). The repeated access of the elements provides temporal locality as well. Parallelism is also readily available since each output element can be computed in parallel. However, the time complexity of this algorithm makes it unsuitable for use on medium to large datasets.

FFT's properties are pretty much the opposite. The loglinear time complexity makes it suitable for essentially any dataset that can fit into RAM. However, there is extremely poor spatial and temporal locality translating to poor cache performance. With each recursive call, the algorithm allocates new memory for the even and odd vectors and then populates them. Therefore, although each recursive call can be parallelized, the usage of memory bandwidth needs to be greatly improved: a significant challenge.

2D FFT, on the other hand, is relatively easy to parallelize (assuming we have a good 1D implementation). Parallelism is readily available across rows and columns. One point to keep in mind is that directly performing 1D FFT on the columns would require accesses with row-length strides which is terrible for spatial locality; this will need to be remedied for a performant implementation.

To resolve these challenges and to improve performance, we apply and explore various optimizations such as iterative formulations, pre-computations, chunking by cache size, matrix transposes, and others.

Approach

Our implementations of the algorithms made use of OpenMP for parallelization. Our measurements during testing were predominantly done on the GHC machines. However, we also used the Bridges 2 supercomputer if tuning parameters was required. Finally, most of our results were measured on the Bridges 2 supercomputer since it provided far more hardware threads than the GHC machines; this allowed us to see how our implementations scale across a large range of thread counts.

DFT

We began our implementation with the $O(n^2)$ DFT implementation described in the Background. We allocated new memory for the result vector since each Fourier coefficient requires all the inputs; note that this is in contrast to our final iterative FFT implementation which is done completely in place. Another way of formulating this algorithm is as a matrix-vector product where the vector is the input vector and the matrix is as below

$$\begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

The algorithm was easily translated into code as a nested loop with the outer loop iterating over the output vector and the inner loop iterating over the inner vector to compute the row-vector dot product. We computed the powers of w_N on the fly in the inner loop using std::polar. Doing so allowed each Fourier coefficient (iteration of outer loop) to be considered as a unit of parallelization, and we simply parallelized this loop with "omp parallel for". Additionally, the scheduling type chosen was static as the computation performed is nearly identical for every output element, which implies that each unit of parallelism takes roughly the same time to compute. The structure of the algorithm allowed for extremely good spatial locality as elements in the original input vector were being accessed (read-only) sequentially, as were the writes to the output vector. The execution time obtained when running on 8 cores for a data set of 2^{15} elements was 2556 ms.

When revisiting this implementation at a later date, we utilized perf to observe that the calls to std::polar in the inner loop were a bottleneck. We observed that only N elements of the matrix actually needed to be computed rather than N^2 since $\omega_N^{k+N} = \omega_N^k$. Therefore, we added an optimization to pre-compute w_N^0, w_N^2, …, w_N^{N-1} and store the results in a vector; since each of these elements can be computed independently using a std::polar call, the pre-computation was trivially parallelizable. We were essentially trading the cost of a std::polar call for an array access with this optimization. This change provided a significant performance improvement. The execution time with the same parameters achieved a result of 1126 ms (a speedup of 2.27x). As can be seen in the perf reports below, there was a significant reduction in the percentage of CPU cycles spent in the trigonometric functions (std::polar is implemented using these).

Before Optimization

```
Samples: 82K of event 'cycles', Event count (approx.): 92045991698
Overhead  Command  Shared Object          Symbol
  55.94%  fft.out  libm.so.6              [.] __atanf
  14.76%  fft.out  fft.out                [.] dft<false>
   9.61%  fft.out  libm.so.6              [.] __asinhf
   8.88%  fft.out  libm.so.6              [.] __gammaf_r_finite@GLIBC_2.15
   5.82%  fft.out  libm.so.6              [.] __cbrtf
   3.12%  fft.out  libm.so.6              [.] __kernel_tanf
   0.56%  fft.out  libm.so.6              [.] __cosf_ifunc
   0.52%  fft.out  fft.out                [.] 0x0000000000002440
   0.35%  fft.out  libgomp.so.1.0.0       [.] 0x000000000002074a
```

After Optimization

```
Samples: 36K of event 'cycles', Event count (approx.): 40501050693
Overhead  Command  Shared Object          Symbol
  99.17%  fft.out  fft.out                [.] dft<false>
   0.67%  fft.out  libgomp.so.1.0.0       [.] 0x000000000002074a
   0.06%  fft.out  libgomp.so.1.0.0       [.] 0x0000000000020592
```

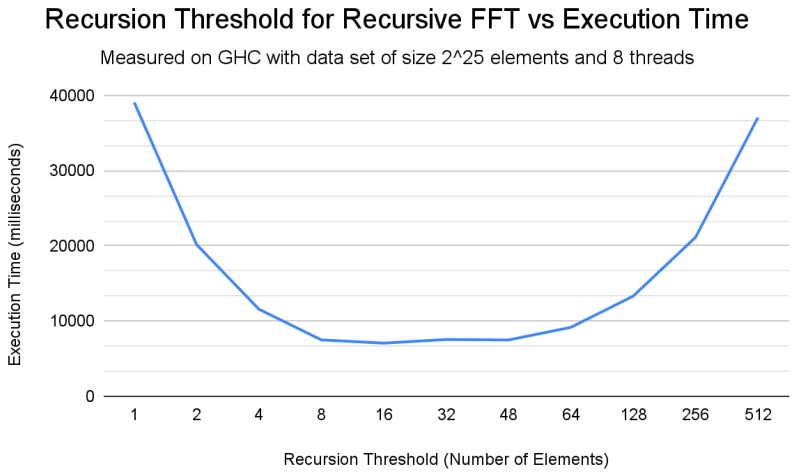
Finally, as we show in the Parallelized DFT section, this implementation scales very well to large thread counts, despite the poor absolute performance on large datasets.

Recursive FFT

Next, we moved our attention to the Cooley-Tukey algorithm for FFT. We implemented this algorithm recursively as described in the Background. Our unit of parallelization was each recursive call, and we parallelized this using OpenMP tasks. However, doing so did not improve performance. In fact, the algorithm showed a slowdown with an increase in the number of threads (see Parallelized FFTs). We hypothesized the cause of this issue to be the memory access patterns, which exhibited little locality. At each recursive call, the algorithm allocates new

vectors for the even and odd elements. This copying of data leads to poor temporal locality. Moreover, allocations themselves aren't cheap. Additionally, creating a new task at high recursion depths made little sense since the overhead of task scheduling likely overwhelms any gains in increased parallelism.

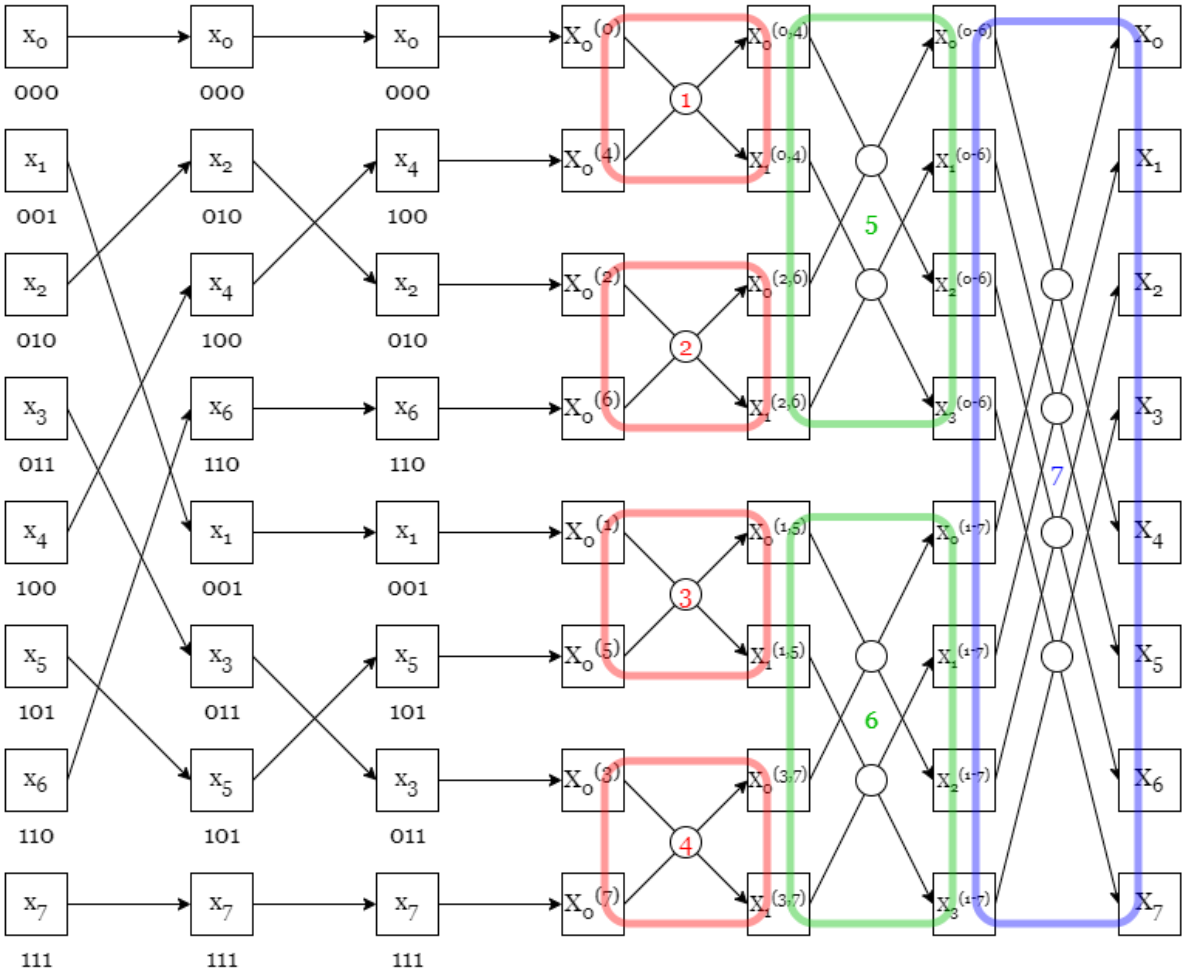
In order to alleviate these issues, we added an optimization that would switch the algorithm to use our DFT algorithm (in serial mode) below a certain size of the input array. This was motivated by the extremely good scaling exhibited by DFT and its access patterns. After tuning the threshold, we found that the best size was 16 on the GHC machines (graph below) and 64 on Bridges 2. This change simultaneously resolved the three issues we mentioned earlier. Performance was slightly worse on a single thread (likely due to the quadratic behavior outweighing other factors), but it was much better at higher thread counts (see Parallelized FFTs). However, since this improvement is only applicable below a certain threshold, the speedup isn't great–especially at high thread counts (see Parallelized FFTs).



It's worth noting that simply not creating more tasks at high recursion depths (using the final keyword in OpenMP) did not scale nearly as well, which further supports the theory that the poor locality was also a key issue.

Iterative FFT

Our next implementation started as an iterative, in-place implementation of the Cooley-Tukey algorithm. This implementation can be "derived" and parallelized by visualizing the operations performed by the recursive implementation. The illustration of this is below, and we will elaborate on it and refer to it (as the butterfly diagram) in the following paragraphs. Note also that we refer to the colored boxes as "sections".



The recursive implementation can be broadly broken down into two parts–a shuffling step where it recursively partitions the even and odd elements, and a bottom-up "combining" step where the now reordered elements are combined using the recursive formulation in Background.

In the first shuffle step, notice how partitioning the even and odd elements is identical to a stable sort by the least significant bit (LSB) of the index. Similarly, the second shuffle step breaks ties with a stable sort by the 2nd most LSB of the original index. Finally, if our sort were unstable, we would need to break ties once again by sorting by the most significant bit (MSB) of the original index. This repeated sorting is identical to sorting the original vector by the *reverse* of the bits of the original *index*. Additionally, since the bit reversal function $bitrev: \mathbb{Z}_N \to \mathbb{Z}_N$ is a bijection, the element at index i in the original vector will be at index bitrev(i) in the final shuffled/sorted vector. Lastly, the elements at index i and bitrev(i) swap places since bitrev(bitrev(i)) = i.

These observations lead to a highly parallelizable algorithm for the shuffle step. Where a is the input vector, simply swap a[i] and a[bitrev(i)]. Each swap is independent of every other swap operation which allows us to parallelize across all of them.

Our remaining optimizations were in the bottom-up step. This can be iteratively implemented as a triple nested loop. The outer loop iterates through the lengths of the combine steps (2, 4, and 8 in our butterfly diagram). The middle loop iterates through every section of a given combine step (sections 1-4 for the first step, sections 5, 6 for the second, and section 7 for the third in the diagram). Finally, the inner loop simply iterates through the elements in every chunk. The outer loop must be done sequentially, so in our initial implementation of the combine step, we parallelized the middle loop. This approach to parallelization essentially exploits the same parallelism used by the recursive implementation, where each recursive call is a unit of parallelization.

Due to the iterative, in-place nature of this implementation, it was already more performant than our optimized recursive implementation. However, more optimizations could be applied to this version. Similar to the precomputation optimization mentioned in DFT, each section in a combine step uses the same roots of unity. A 2-point FFT (section 1) requires w_2^0, a 4-point FFT (sections 1, 2, 5 together) requires w_4^0, w_4^1 and the roots required by a 2-point FFT, and so on. In general, an N-point FFT requires the following roots

$$\omega_2^0, \omega_4^0, \omega_4^1, \ldots, \omega_N^0, \omega_N^1, \ldots, \omega_N^{\frac{N}{2}-1}$$

This can be implemented as a nested loop where the outer loop iterates over the subscripts and the inner loop iterates over the superscripts/exponents. Precomputation greatly improved performance; on a single thread, we saw a 2.2x improvement. However, we noticed that this step quickly became a serial bottleneck, with speedup on 4 and 8 threads being nearly identical with this optimization. OpenMP doesn't allow us to directly parallelize the outer loop due to the non-standard loop increment of doubling the loop variable. Parallelizing just the inner loop actually leads to a slowdown: likely due to false sharing. Moreover, even if we were able to parallelize the outer loop, it wouldn't be performant due to the uneven workload and limited number of total iterations. The computation of every element in the precomputation vector is independent of every other element, so we would ideally be exploiting parallelism across all the elements. To achieve this, we manually collapsed the nested loop as follows

Before

```
for (int32_t len = 2; len <= n; len *= 2) {
  for (int32_t i = 0; i < len / 2; ++i) {
```

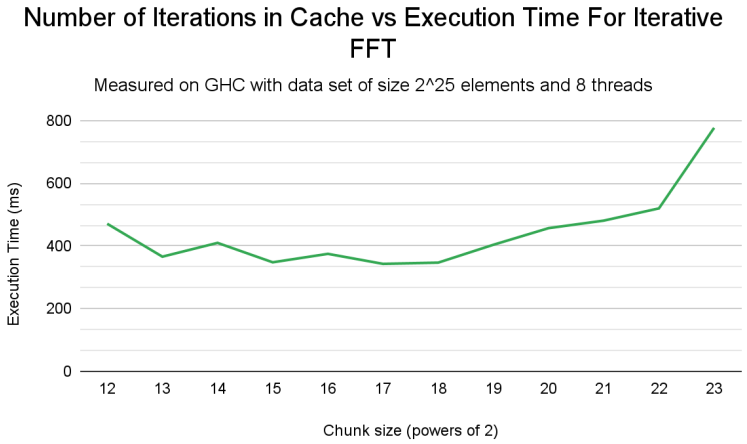
After

```
for (int32_t j = 1; j < n; ++j) {
  int32_t temp = 1 << floor_log2(j);
  int32_t len = 2 * temp;
  int32_t i = j - temp;
```

Now, the j-loop is trivially parallelizable.

Our next optimization improved the access patterns of the bottom-up step by leveraging data reuse and improving temporal locality. This optimization applies to both sequential and parallel execution, so we will describe it in terms of sequential execution for simplicity. Consider the butterfly diagram above and a machine where 4 elements fit in the L1/L2 cache. The algorithm executes the sections in the order 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7. Executing sections 1 and 2 would bring in the memory corresponding to the first half of the array into the cache. Notice that section 5 accesses this same chunk of memory. However, by the time we execute section 5, the first half of the array is no longer in the cache since sections 3 and 4 would have replaced the cache with the second half of the array. The same problem occurs for section 6. If we reorder the execution of sections to 1 -> 2 -> 5 -> 3 -> 4 -> 6 -> 7, sections 5 and 6 will operate on data that is already present in the cache and won't need to go out to main memory. With this observation, we can split the bottom-up step into 2 parts. A "chunking" step where we sequentially execute sections that operate on data that fits in the cache (sections 1, 2, 5 form a chunk and so do 3, 4, 6) followed by a "non-chunking" step where we execute any remaining combine steps that operate on data too large to fit in the cache (section 7 in our example).

The bottom-up step previously required transferring $N \log_2 N$ elements from memory ($\log_2 N$ combine steps). With this optimization, the combine steps that fit in the cache only require transferring N elements in total, and there are $\log_2 C$ such steps where C is the number of elements that fit in the cache. The total number of element transfers with this optimization is then $N \log_2 N - N \log_2 C + N = N (\log_2 \frac{N}{C} + 1)$. We show in Parallelized FFTs that the "chunking" step scales significantly better than the "non-chunking" step. This optimization is parameterized by the number of iterations in a chunk. The parameter should roughly correspond to how many std::complex<double> elements fit in the L1/L2 cache. Since each complex is 16 bytes, our back-of-the-napkin calculations yielded around 2^{15}. Empirical measurements matched this result as shown below.



Finally, we noticed that we weren't exploiting all the parallelism that was available in the bottom-up step. While we parallelized across sections (like in the recursive implementation), we weren't parallelizing within a section. This would mainly be an issue at the later combine steps. For example, on a 2-thread machine, section 7 of the butterfly diagram would be executed by a single thread which is a bottleneck. We were able to exploit the parallelism in these steps as well by collapsing the middle and inner loops of our implementation as follows (note that this can also be done using the "collapse" OpenMP keyword with slightly worse performance)

Before

```
for (int32_t j = 0; j < n; j += len) {
  for (int32_t i = 0; i < len / 2; ++i) {
```

After

```
for (int32_t k = 0; k < n; k += 2) {
  int32_t j = k - (k & (len - 1));
  int32_t i = (k - j) / 2;
```

The final algorithm can be split into the following components

1. Bit reversed sort
2. Precomputation of roots
3. Chunking: Bottom-up steps that fit in the cache
4. Non-chunking: Bottom-up steps that don't fit in the cache

2D FFT

Our implementation for this was relatively simple as we were able to use our iterative FFT implementation as a subroutine. The main challenge was the strided access when performing 1D FFTs on the columns (see Background for details). We were able to mitigate this using matrix transposes. The algorithm looked as follows

1. Perform in-place 1D FFT on the rows
2. Transpose the matrix
3. Perform in-place 1D FFT on the rows (which were the columns pre-transpose)
4. Transpose the matrix

The transpose was done parallelly and in-place using a nested loop. Parallelizing just the outer loop wasn't ideal due to the "triangular" work distribution of the loop iterations. We were able to parallelize both loops using the "collapse(n)" OpenMP keyword, which indicates that parallelism is available in the subsequent n nested loops starting from the outermost (collapse(1) being the default).

We were then left with a choice of either exploiting parallelism across rows and columns or within rows and columns. In other words, we could either parallelly compute several 1D FFTs (using our 1D FFT implementation in serial mode), or we could parallelly compute the 1D FFT but only one at a time. The latter would be better if loading the rows into memory is the bottleneck since parallelizing across rows wouldn't help in that case, while the former would be

better otherwise since it exploits better parallelism. We found that the former scaled significantly better than the latter, and we explain this further in Parallelized 2D FFT.

Results

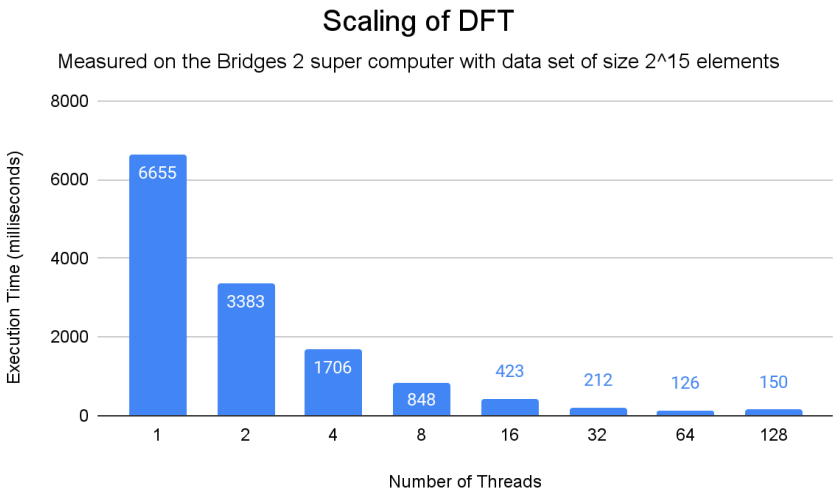
Introduction

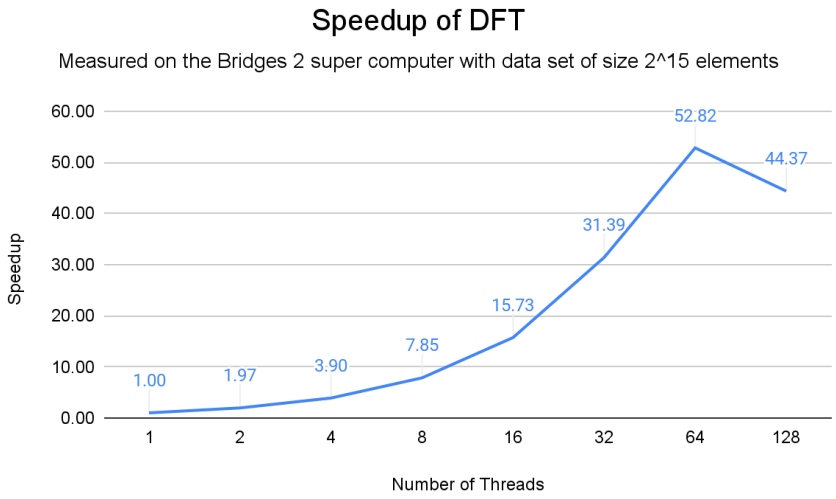
We believe that we were successful in achieving our goals. Not only were we able to effectively parallelize the Fourier transforms (DFT, 1D FFT, and 2D FFT), but we were also able to achieve good speedup with optimizations and mitigating bandwidth bottlenecks. Additionally, we were able to visualize the correctness of our implementations with an image compression tool. We note that our most performant 1D FFT algorithm was our optimized, iterative version as we had expected at the start of the project.

For our project, we measure our performance in terms of execution time as well as speedup. Execution time allowed us to compare performance between our implementations, while speedup allowed us to see how our implementations scale. It is worth noting here that we didn't sacrifice absolute performance in exchange for speedup; for example, eliminating precomputation would make our iterative FFT implementation more compute bound and scale better, but its absolute performance would still be worse.

To test our 1D implementations, we predominantly used 5 randomly generated datasets with 2^{10}, 2^{12}, 2^{15}, 2^{20}, and 2^{25} elements. 2^{15} was the limit beyond which the quadratic implementation became infeasible to use. Therefore, we decided to reserve the larger datasets (2^{20}, 2^{25}) for the FFT implementations. For our 2D implementation, we used randomly generated datasets with $2^{10} \times 2^{10}$, $2^{11} \times 2^{11}$, $2^{12} \times 2^{12}$, $2^{13} \times 2^{13}$, and $2^{14} \times 2^{14}$ elements.

Parallelized DFT

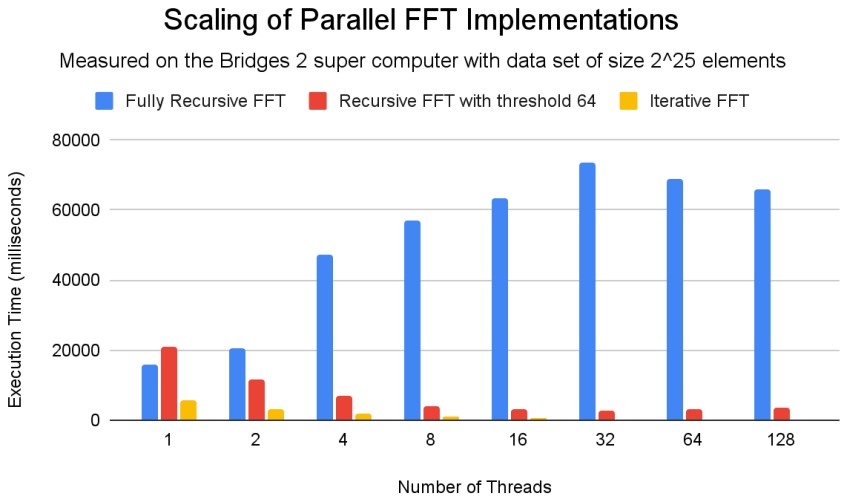


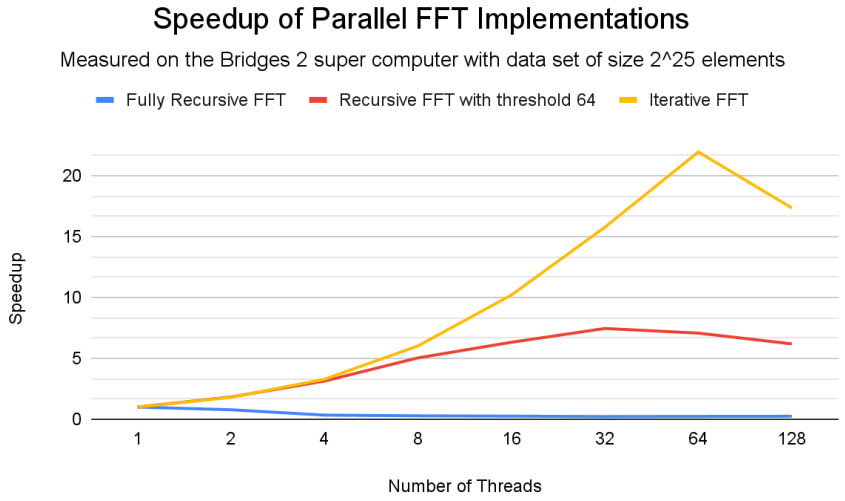


From the above two charts, we can observe that the DFT implementation scales extremely well (linear up to 32 and nearly linear up to 64). However, its absolute performance relative to the FFT implementations is extremely poor. This poor performance can be explained by the quadratic time complexity of this implementation. The good scaling can be explained by the access patterns of the algorithm–specifically, the good spatial and temporal locality. By accessing consecutive elements, the majority of the reads will be cache hits. Additionally, the dataset size also likely played a role since 2^{15} elements can fit in the higher level caches of the Bridges 2 nodes at least.

However, we notice that there is a decrease in speedup and performance when increasing the number of threads from 64 to 128; this is a recurring theme in almost all our measurements. We suspect that this is because the Bridges 2 supercomputer has a 2 CPU architecture (2 x 64 cores) with an interconnect connecting the CPUs. At 128 threads, we are significantly utilizing the interconnect between the 2 CPUs and this data movement can become a bottleneck that isn't as prevalent at lower thread counts. Partial utilization of the interconnect at 64 threads can also explain the lack of linear speedup at the thread count. We believe this is the case as we observed that not all of the 64 threads were scheduled on a single CPU. We didn't observe this behavior on the GHC machines.

Parallelized FFTs



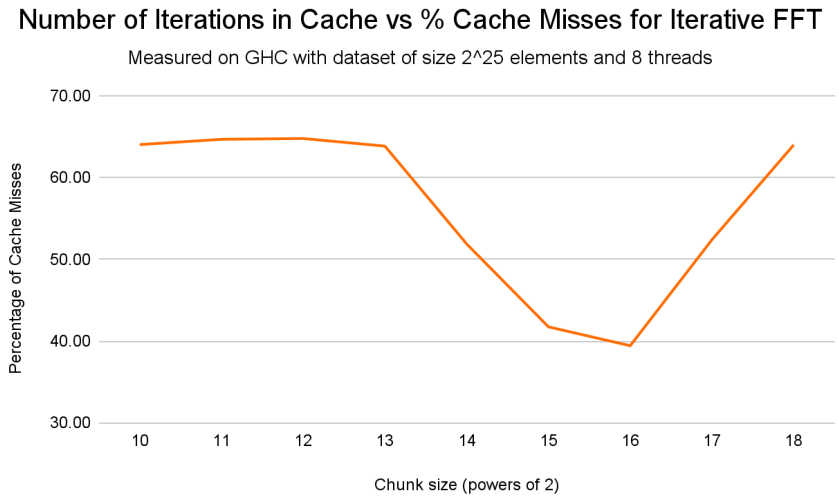


The above two graphs show the relative performance of our 1D FFT implementations. Clearly, the naive parallelization of the recursive implementation is terrible. Not only is it slower than the other implementations, but it also has terrible scaling (in fact it slows down in some cases when the number of threads is increased). The poor absolute performance is expected as it has bad locality. As explained in Recursive FFT, this implementation partitions an array into 2 newly allocated arrays; this copying back and forth between arrays causes extremely poor locality. Some of the poor scaling can be attributed to the fixed costs associated with creating tasks and the synchronization overhead that OpenMP would introduce at higher thread counts due to concurrent task retrieval. Another possible factor is that the recursive calls for small arrays are assigned to different threads; if these arrays are laid out closely in memory (due to allocator

behavior), there is the possibility of false sharing between threads as well. This can help explain the slowdown we get when increasing thread count.

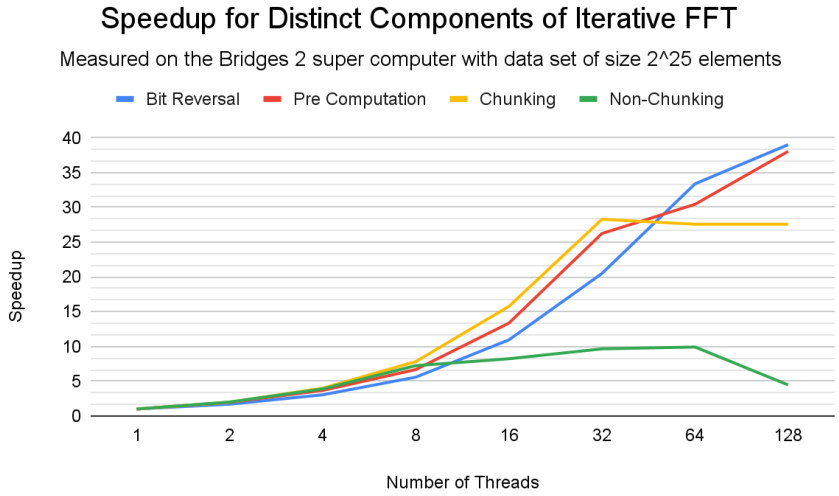
We also see that switching to the quadratic DFT implementation at a threshold of 64 performs better in terms of overall execution time as well as speedup compared to the fully recursive implementation. As we explained in Recursive FFT, this is due to a combination of not creating new tasks at small array sizes and the better access patterns of the DFT implementation. With this optimization, we at least get a speedup with an increase in threads unlike our initial implementation.

Lastly, the iterative FFT implementation has the best performance across all the 1D implementations in terms of absolute performance as well as speedup. There are several factors that explain this observation. One of which is the use of static scheduling rather than dynamic tasks unlike the recursive implementation. We explain all the optimizations in this implementation in Iterative FFT (with measurements). The use of chunking reduces the amount of data that needs to be transferred from RAM due to improved temporal locality. We can verify this benefit empirically in the graph below, which shows that as we decrease or increase the chunk size from its optimum, we observe an increase in the percentage of cache misses (decrease in performance).



It is important to note that the speedup achieved for iterative FFT is not linear relative to the number of threads. At lower thread counts (1, 2, 4, 8), the speedup is almost linear. However, at higher thread counts we see that this is not the case. We can attribute most of this due to poor speedup in the non-chunking section of our algorithm as we show below. This section is the most bandwidth bound and as we increase the number of threads, the bandwidth between RAM and the CPU stays constant while the amount of compute available increases. We discuss this in further detail below. However, as we noted in Introduction, speedup alone isn't the best metric as it would have been relatively easy to sacrifice absolute performance in exchange for speedup by

eliminating precomputation, for example, and having threads do repeated work. This was not our goal, as we were aiming to maximize absolute performance, and this is reflected by the fact that our iterative implementation, at 64 threads, is 260x faster than our baseline recursive implementation and 11.2x faster than our optimized recursive implementation.

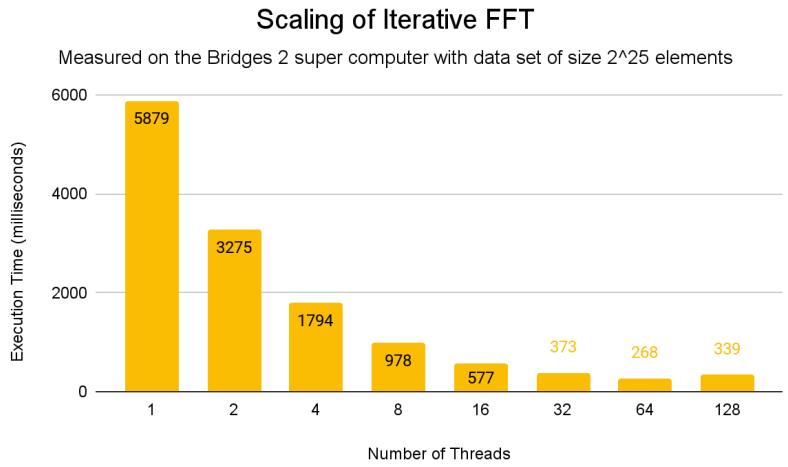


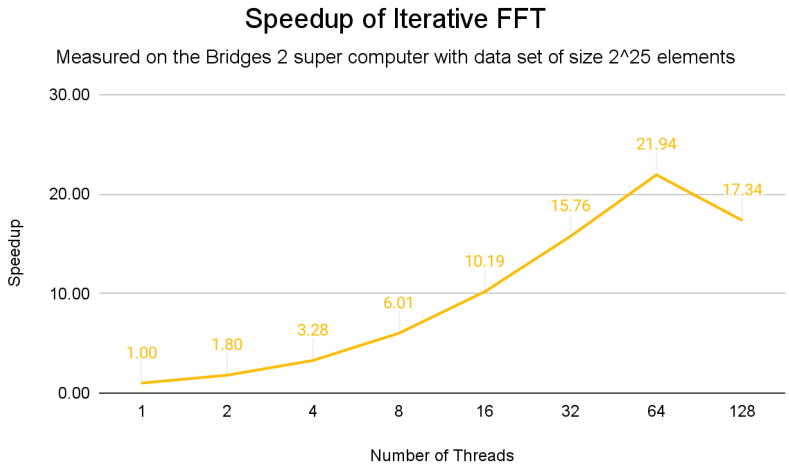
The iterative FFT algorithm can be split into four components as described in Iterative FFT: bit reversal, pre-computation, chunking, and non-chunking. The above graph shows the speedup of each of these components. As can be observed, the non-chunking component performs the worst. This is expected as this component performs the FFT calculation across a set of elements that do not fit in the cache causing this to be the most bandwidth bound. On the other hand, this also illustrates the effectiveness of the chunking optimization as this component has relatively good speedup. Although at high thread counts, it seems to exhaust the bandwidth as well.

The bit reversal and precomputation components scale the best. This is expected since they both perform computationally expensive std::polar calls as well as other compute-bound arithmetic. However, although they are maximally parallelized, they begin to be limited by available bandwidth at high thread counts.

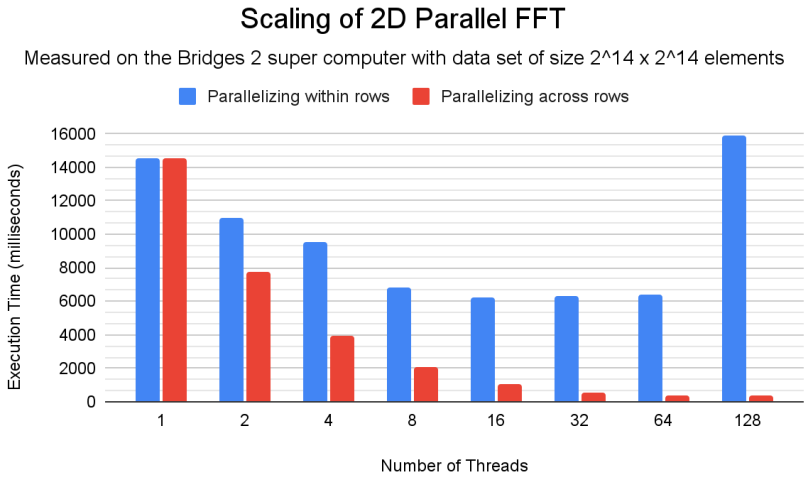
Lastly, the increase in execution time and decrease in speedup when increasing the number of threads from 64 to 128 across the 1D FFT implementations can be attributed to the CPU architecture of the Bridges 2 supercomputer as we observed in Parallelized DFT as well.

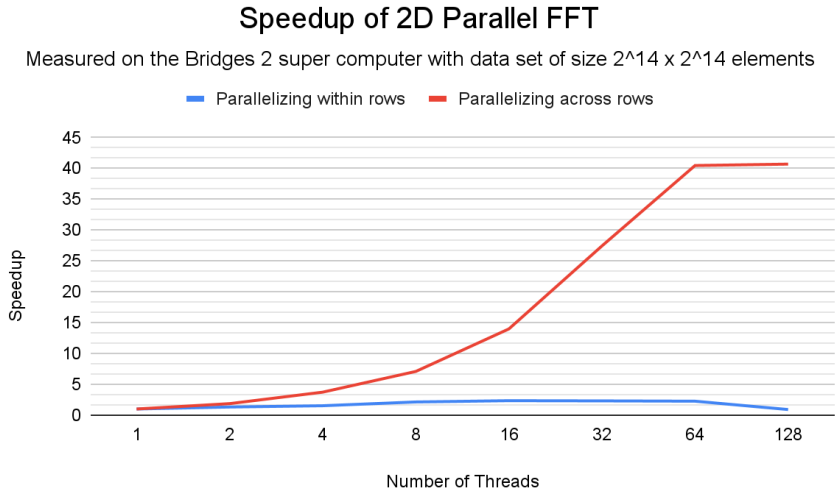
Below, we have provided more granular graphs of just our iterative implementation.





Parallelized 2D FFT



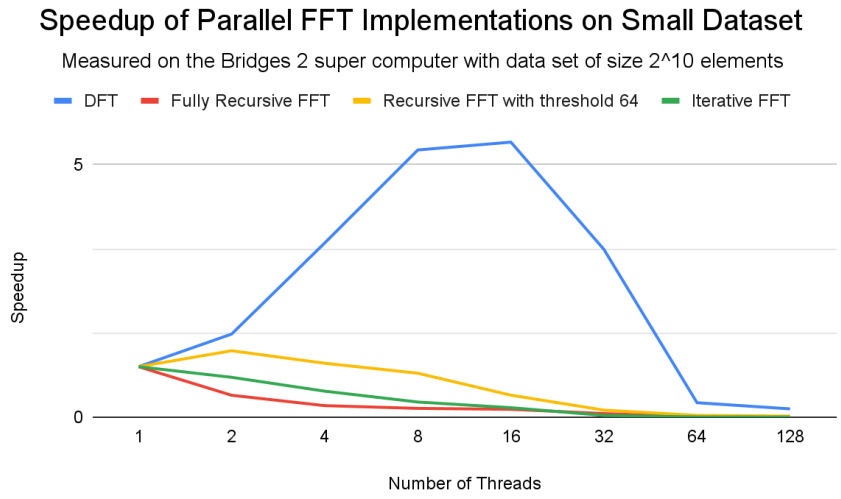


As noted in 2D FFT, we had a choice of either parallelizing across rows or within rows for our implementation. We found that the former approach was better than the latter at least for our dataset sizes where the rows are relatively small (since the entire matrix still needs to fit in memory). All our datasets had rows small enough to fit in the L1d cache of GHC machines (256 KB). Some datasets had rows that fit in the L1d cache of Bridges 2 nodes (32 KB), while all fit in the L2 cache (512 KB) of these machines. Due to this, each 1D FFT was compute bound since it didn't repeatedly need to go to main memory regardless of how it was accessing the data since the rows didn't need to be evicted.

Our results reflect this observation, and we see that we attain nearly linear speedup up to 64 threads (40x speedup on 64 threads). We suspect the plateau in performance at 128 threads is due to increased use of the interconnect between the 2 chips on the Bridges 2 nodes as observed in Parallelized DFT.

Miscellaneous Observations

We note that smaller data sets exhibit poor performance as execution time is dominated by overheads in our implementations. For example, when using a data set of size 2^{10}, we observe that increasing the number of threads results in a slowdown and decrease in execution time. Furthermore, when considering our iterative FFT implementation, the chunking optimization is irrelevant as a small dataset may fit entirely in the cache, and we end up not exploiting any parallelism. Lastly, fixed costs such as thread creation and context switches may dominate execution time for small datasets. The graph below shows the performance of the 1D implementations when running on a small dataset to support the observations made. In this instance, it is expected for DFT to have the best scaling due to the extremely low overhead.



Lastly, considering that our target application of image compression is usually performed on CPUs, we believe that our choice of a CPU-focused implementation, as well as our testing on the GHC and Bridges 2 machines, were accurate and reflective of the real-world applications of this algorithm. However, it is worth noting that FFT can be performed on GPUs as well; in fact, CUDA provides the cuFFT library for this purpose.

Image Compression

We can apply FFT to an image by treating each color channel in an image as a 2D matrix of elements. 2D FFT can then be applied to this matrix to yield the frequencies corresponding to each color channel. FFT, as an algorithm for image compression, relies on the fact that most of the information in a real-world image is present in a few low frequencies while little information is carried in the higher frequencies. To (lossily) compress an image, we can discard some percentage of the Fourier coefficients with the lowest magnitude and only store the remaining coefficients and their respective indices. To reconstruct the image, we simply recreate the Fourier coefficient matrix (filling in 0s for the ones that were discarded) and apply the inverse Fourier transform.

Below we have included an image (Mona Lisa) that we have compressed at 5 different levels (50%, 75%, 90%, 98%, and 99.5%). The compression was done using our parallel 2D FFT implementation. The accurate results showcase the correctness of our implementations as well as a real-life, intriguing use case of FFT.

Compression ratios in row-major order: 0% (Original), 50%, 75%, 90%, 98%, 99.5%

References

Akin, B., Franchetti, F., & Hoe, J. C. (2015). FFTs with Near-Optimal Memory Access Through Block Data Layouts: Algorithm, Architecture and Design Automation. *Journal of Signal Processing Systems*, *85*(1), 67–82. https://doi.org/10.1007/s11265-015-1018-0

Bader, M. (2018). *Algorithms of Scientific Computing Fast Fourier Transform (FFT)*. https://www5.in.tum.de/lehre/vorlesungen/asc/ss15/fft.pdf

Brunton, S. (2020a). *Image Compression and the FFT (Examples in Python)*. Youtube. https://www.youtube.com/watch?v=uB3v6n8t2dQ

Brunton, S. (2020b). *The Discrete Fourier Transform (DFT)*. Youtube. https://www.youtube.com/watch?v=nl9TZanwbBk

Brunton, S. (2020c). The Fast Fourier Transform Algorithm. Youtube. https://www.youtube.com/watch?v=toj_IoCQE-4

Popovici, D. T., Low, T. M., & Franchetti, F. (2018). Large Bandwidth-Efficient FFTs on Multicore and Multi-socket Systems. *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. https://doi.org/10.1109/ipdps.2018.00048

Wikipedia Contributors. (2019a, January 8). *Discrete Fourier transform*. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/Discrete_Fourier_transform

Wikipedia Contributors. (2019b, September 17). *Cooley–Tukey FFT algorithm*. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm

Wikipedia Contributors. (2022, October 18). *DFT matrix*. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/DFT_matrix