Investigating Communication Patterns for Distributed Fast Fourier Transforms

Dr. Afrin Naz, West Virginia University Institute of Technology.

Dr. Afrin Naz is an assistant professor at the Computer Science and Information Systems department at West Virginia University Institute of Technology. She is working with high school teachers to inspire the K-12 students to the STEM fields. In last four years Dr. Naz and her team launched six workshops for high school teachers. Currently her team is training the high school teachers to offer online materials to supplement their face-to-face classroom.

Mardigon Max Toler, West Virginia University Institute of Technology

Mardigon Toler is a student of Computer Science and Mathematics at West Virginia University Institute of Technology, finishing a bachelor's degree in both fields in spring 2019. His interests include digital audio, digital signal processing, and distributed and parallel computing. His past projects have included applications of AI to real-time music accompaniment as well as real-time software-based audio synthesis using Fourier transforms.

Investigating Communication Patterns for Distributed Fast Fourier Transforms

Introduction

Fast Fourier Transforms are efficient signal processing algorithms for performing frequency analysis on (or synthesis of) signals by computing the signal's Discrete Fourier Transform. In this paper, we describe two methods for parallelizing one-dimensional digital signals in distributed memory computer clusters. The first method consists of assigning multiple processes to different portions of the input signal and having half of the active processes send their results to the remaining half during each successive stage. This communication pattern is easy to implement but has the disadvantage that its scope of parallelism decreases during the operation of the algorithm. A second algorithm is presented and described in which all processes remain active throughout. Advantages and disadvantages of these two alternative patterns are explored, along with ideas for improving the latter algorithm. Some data is collected on a small cluster of inexpensive consumer-grade hardware to explore the feasibility of this algorithm.

Context

The Fast Fourier Transform (FFT) is an algorithm for computing the Discrete Fourier Transform (DFT) of a sequence of samples of a signal. The DFT of a signal in time or space is a representation of that signal in the frequency domain. The DFT is a useful tool in digital signal processing because it describes how a digital signal is made up of complex sinusoidal components. The 1-dimensional DFT of a signal y is defined as

$$Y_k = \sum_{n=0}^{N-1} y_n e^{-i2\pi kn/N}$$

where N is the number of samples in the signal being considered. When y is an N-point signal, the DFT of y is also an N-point signal. By Euler's Formula

$$e^{ix} = \cos(x) + i\sin(x)$$

we can alternatively define the DFT as

$$Y_k = \sum_{n=0}^{N-1} y_n \left(\cos\left(\frac{2\pi k}{N} n\right) - i \sin\left(\frac{2\pi k}{N} n\right)\right)$$

As we can see, the DFT is a complex valued sequence of values that are derived from multiplying each sample of the input signal with samples of complex sinusoids of varying frequencies. The result, Y, is able to represent the how much a sinusoid of each of these frequencies contributed to the input signal, as well as its phase [1][2].

Fast Fourier Transforms are a more efficient way of computing the DFT. The Cooley-Tukey FFT algorithm takes advantage of the possibility of reusing intermediate calculations during the computation of the DFT [2]. The sum in the DFT can be decomposed into its even and odd terms, and we can use properties of the complex exponential function to rewrite it as in the following derivation.

$$Y_k = \sum_{n=0}^{N-1} y_n e^{-i2\pi kn/N}$$

$$= \left(\sum_{n=0}^{\frac{N}{2}-1} y_{(2n)} e^{-i2\pi k(2n)/N}\right) + \left(\sum_{n=0}^{\frac{N}{2}-1} y_{(2n+1)} e^{-i2\pi k(2n+1)/N}\right)$$

$$= \left(\sum_{n=0}^{\frac{N}{2}-1} y_{(2n)} e^{-i2\pi kn\frac{2}{N}}\right) + \left(\sum_{n=0}^{\frac{N}{2}-1} y_{(2n+1)} e^{(-i2\pi k2n/N)+(-i2\pi k/N)}\right)$$

$$= \left(\sum_{n=0}^{\frac{N}{2}-1} y_{(2n)} e^{(-i2\pi kn)/\frac{N}{2}}\right) + \left(\sum_{n=0}^{\frac{N}{2}-1} y_{(2n+1)} e^{-i2\pi kn/\frac{N}{2}} \cdot e^{-i2\pi k/N}\right)$$

$$= \left(\sum_{n=0}^{\frac{N}{2}-1} y_{(2n)} e^{(-i2\pi kn)/\frac{N}{2}}\right) + e^{-i2\pi k/N} \cdot \left(\sum_{n=0}^{\frac{N}{2}-1} y_{(2n+1)} e^{-i2\pi kn/\frac{N}{2}}\right)$$

So, we see that a DFT of size N can be found by calculating two DFTs of size N/2 and adjusting by multiplying with a complex sinusoid. Those two N/2 sized DFTs can again be decomposed the same way. This version of the FFT involves rearranging the n input samples and then, over *log(N)* stages, involving pairs of these samples in "butterfly operations" involving multiplying them with samples of a complex sinusoid [1]. After the final stage, the input array will have been replaced with its DFT. FFT algorithms are capable of being parallelized to a high degree in computers by using distributed-memory, shared-memory, or a mixture of both. Although parallelization is perhaps most effective in multi-dimensional FFTs, it is still possible in the one-dimensional case.

We will examine two possibilities for parallelizing the one-dimensional FFT algorithm in the context of a distributed-memory computing cluster. Data is presented from applying these ideas on a small cluster of four cheap single-board computers (Rockchip Quad-Core ARM Cortex-A17 RK3288 processors, 1.8GHz, 32KB L1 cache, 1MB L2 cache) connected with gigabit network cards on a gigabit Ethernet network. Processes are spawned equally across the network and collaborate with message passing using MPI. In all cases, n will be a power of two, and the number of processes will also be a power of two. It is not expected that the performance of these programs on this cluster of inexpensive hardware is representative of clusters found in professional applications, and no statistical tests were performed

Algorithms

A very common, easily implementable way to parallelize the FFT is to distribute all samples of the input evenly across all processes. [3] describes this pattern of communication and presents results for its performance compared with some alternative methods. In early first stage of the FFT, the processes can perform butterfly operations on the samples assigned to them. For the next stage, half of the processes will have samples that need to be involved with butterfly operations with samples in the memory of process from the *other* half of the processes. So, after the initial stage is complete, half of the processes pass their data to the other half and then deactivate. The deactivated processes do not ever participate in the algorithm past this stage. At the end of every stage, half of the remaining processes will deactivate, until finally the last stage occurs in which all data and operations are in only one process. With this communication pattern, the algorithm has a degree of parallelization that is variable across different stages. Many of the processes do not contribute to most of the calculation of the DFT, and in later stages, the load on some processors begins to increase as they become responsible for more and more data.

However, it *is* possible to devise a pattern of communication such that every process remains active and participant during every stage of a parallel FFT. This solution would achieve a uniform degree of parallelization at the cost of a more complicated pattern of communication and possibly more communication overhead for message passing. Instead of processes merging their samples into another process's memory and deactivating, they instead send half of their samples to another process and receive an equal number of samples from a different process. The point is to ensure that a process receives the samples that need to interact with its retained samples in butterfly operations.

Figure 1 shows an example of the communication and operations that would occur during this algorithm with n=16. A square in each stage represents one sample of the input sequence. The samples of this buffer are, in practice, distributed evenly across all processes. Each process allocates enough memory for the entire buffer to ease calculation of indices and reasoning about the algorithm, but this could be further optimized. In this figure, solid lines represent a butterfly operation, while dashed lines represent movement of samples to a different location in the buffer, either through local copying or through MPI operations to another process's memory.

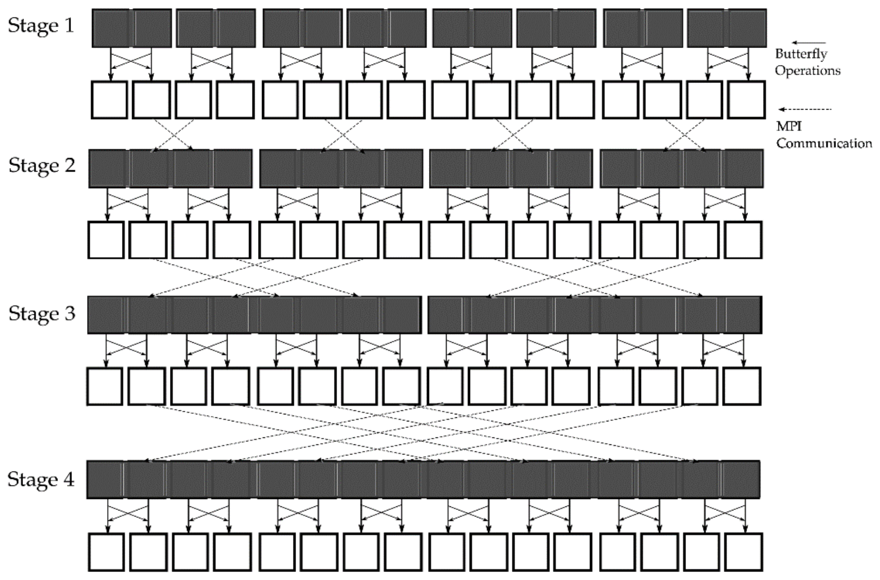

Figure. 1 Example of data communication and organization

During each stage, butterfly operations will always be between adjacent points in the buffer. Each pair of samples can be called a group. The buffer is logically divided at each stage into "super groups," which are collections of groups. During the MPI communication phase at the end of each FFT stage, adjacent super groups are interleaved with each other, as shown in Figure 1. At the end of the last stage, all samples will need to be communicated back to the root process that initially distributed all of the data if necessary. Note that the data will be out of order and must be rearranged after (or during) being copied back to the root process.

To achieve this, it is necessary to calculate the indices and process rank that each process needs to send its samples to during each stage. A process must also be able to receive samples from the correct sender. Fortunately, this pattern does not depend on the actual input data; it only relies on n. Therefore, a lookup table can be pre-computed and loaded at runtime. We first define an algorithm in Python-based pseudocode that determines where samples should reside at each stage of the algorithm, as in Figure 2.

```
Algorithm 1 Determining Sample Indices
  function $SimulateInterleaving(n)$:
      $A \leftarrow [0]_{log_2(n) \times n}$
      $A[0] \leftarrow arange(0, n)$
      $superGroups \leftarrow A[0]$
      $sgSize \leftarrow 2$
      $M \leftarrow \lfloor log_2(n) \rfloor$
      for $stage$ in $range(1, M + 1)$ do:
          $row \leftarrow []$
          for $s$ in $range(1, n/sgSize, stepby2)$ do:
              for $g$ in $range(0, sgSize, stepby2)$ do;
                  $row.append(superGroups[(s - 1)(sgSize) + g])$
                  $row.append(superGroups[(s)(sgSize) + g])$
              end for
              for $g$ in $range(0, sgSize, stepby2)$ do;
                  $row.append(superGroups[(s - 1)(sgSize) + g + 1])$
                  $row.append(superGroups[(s)(sgSize) + g + 1])$
              end for
          end for
          $A[stage - 1] \leftarrow row$
          $superGroups \leftarrow row$
          $sgSize \leftarrow 2 \cdot sgSize$
      end for
       return $A$
  end function
```

Figure. 2 Algorithm for computing table A of sample indices at each stage

From this table, we can compute another table that explicitly states the *destination index* that a sample needs to be sent to. That is, using the new table **B**, calculated using the algorithm in Figure 3, during stage *s*, sample *i* in the current buffer should be sent to position **B**[*s*][*i*].

Algorithm 2 Determining a matrix for sample destinations

```
function $SampleDestinations(A, n)$
    $B \leftarrow [0]_{log_2(n)-1 \times n}$
    for $i$ in $range(1, log_2(n))$ do:
        for $j$ in $range(0, n)$ do:
            for $k$ in $range(0, n)$ do:
                if $A[i][k] == A[i-1][j]$ then:
                    $B[i-1][j] \leftarrow k$
                end if
            end for
        end for
    end for
    return $B$
end function
```

Figure. 3 Algorithm for determining table B, which contains the destination of the current stage's samples

Strided vector types in MPI can be used for efficiently communicating non-contiguous samples with only a single call to MPI_Send [4]. Note, however, that in the initial stages of the algorithm, before communication is necessary, samples don't need to be communicated. So, we can rearrange those locally without using MPI calls, and then use table **B** once inter-process communication becomes necessary.

A favorable outcome of this scheme is that all communication that any pair of processes communicating with each other in a stage of the algorithm will not need to communicate with any other processes. This makes synchronization trivial. However, this pattern of communication also introduces a problem: the rearranging of samples during communication cannot be performed in place because some samples may be overwritten before they are communicated to their destination. Note that this issue is not present in the other variant of a parallel FFT and only exists after making this experimental modification to the communication pattern.

There are multiple solutions to the issue. One easy solution is to simply allocate a second buffer for the signal on each process. During one stage, one of these logical buffers will be sent to the other logical buffer. Then, the recipient buffer will become the sending buffer during the next stage. The disadvantages to this approach, however, is that twice as much memory must be allocated, and the approach might be devastating for cache performance. In late stages of the algorithm, some input samples will be received with MPI, processed in a butterfly operation, and then immediately sent to another process's memory. This constant moving of data in and out of the processes' memory space eliminates the opportunity to benefit from keeping data low in the memory hierarchy.

Another approach would be to leverage the extra space that is already allocated. In this algorithm, all processes allocate space for n samples, even though during each stage a process is only ever responsible for computations involving n/z of the samples, where z is the number of processes. Consequently, each process has at least enough space for receiving n/2 processes. So, it is possible to send samples without risking overwriting a sample before it has a chance to be sent (since the program could use the space for those extra n/2 samples for all received samples), but this would violate assumptions in the above algorithms for determining the destinations for each sample at each stage. However, it would be possible to tweak those algorithms to account for using the extra allocated space when determining sample locations.

Testing

Some tests were performed to examine the extent of slow-down induced by the modified scheme for performing this FFT while keeping the work balanced among all processes during all stages. Data is collected for input problems sized at 2^m with m = 8, 9, ..., 13. For each problem size, the time taken to complete the FFT was recorded (using MPI_Wtime) for both algorithms. The signal to be processed was randomized before each measurement.

Results

Table 1 – Time(s) for Variably Parallel FFTs

	Size of Input Sequence					
Trial	**2^8**	**2^9**	**2^10**	**2^11**	**2^12**	**2^13**
0	0.00730491	0.007431984	0.007926941	0.006924868	0.007750034	0.009698868
1	0.07133007	0.071313858	0.04351902	0.013051033	0.003162146	0.004154205
2	0.03854990	0.001168013	0.002104044	0.002905846	0.003079176	0.003484964
3	0.07120109	0.099003077	0.067127943	0.002258062	0.003064871	0.002954006
4	0.03851008	0.001163006	0.002605915	0.007601023	0.00303793	0.003409863
5	0.04117298	0.066781044	0.036784887	0.002853155	0.003007889	0.003483057
6	0.00181007	0.069792032	0.001973152	0.002262831	0.002007008	0.003461123
7	0.06818318	0.069838047	0.037243128	0.002743006	0.002999067	0.003462076
8	0.00186586	0.069776058	0.002100229	0.00275898	0.002447128	0.003501892
9	0.06618714	0.069787025	0.037121058	0.002216101	0.002918959	0.004178047
10	0.06988001	0.069828033	0.002166033	0.002208948	0.002432108	0.003722906
11	0.06985283	0.069777012	0.037161827	0.002761841	0.002968073	0.002723932
12	0.06990814	0.06979394	0.002072096	0.002812862	0.002377033	0.002927065
13	0.03963590	0.069829941	0.036431789	0.002746105	0.002990007	0.003406048
14	0.06963706	0.069765091	0.039648056	0.002213955	0.00241518	0.002942085
15	0.06994104	0.069797993	0.039701223	0.002708912	0.002932072	0.003452063
16	0.06982207	0.069854975	0.039593935	0.002712965	0.002570868	0.003490925
17	0.06985497	0.069751024	0.039649963	0.002701044	0.003030062	0.003483057
18	0.06996107	0.069789171	0.039659977	0.002733946	0.002417088	0.003479958
19	0.06980300	0.069841862	0.039649963	0.002705097	0.003015995	0.003808022
20	0.06985712	0.069743872	0.002003908	0.002722025	0.002824783	0.00355196
21	0.06994915	0.069803953	0.038138151	0.002757072	0.002460957	0.002979994
22	0.06980300	0.069832087	0.002099037	0.002177	0.002427101	0.002853155
23	0.06989813	0.069780111	0.037167788	0.002722025	0.002985001	0.002868891
24	0.06991506	0.069793224	0.002062082	0.002695084	0.002968788	0.003398895
25	0.06980085	0.06983304	0.037271023	0.002158165	0.002403975	0.003415823
26	0.06990099	0.069769144	0.002050877	0.002693176	0.002967119	0.003442049
27	0.06992602	0.069809914	0.037245035	0.002707005	0.002974033	0.003482103
28	0.06978893	0.069830894	0.002064943	0.002163172	0.002886057	0.002927065
29	0.07150793	0.071377039	0.034326077	0.00224185	0.003145933	0.003787041
	Averages					
	0.05915862	0.06412188	0.02495567	0.00323057	0.00295555	0.00359770

Table 2 – Time (s) for Uniformly Parallel FFTs

	Size of Input Sequence

Trial	**2^8**	**2^9**	**2^10**	**2^11**	**2^12**	**2^13**
0	0.048190832	0.034527063	0.072045803	0.02786994	0.025142908	0.007902145
1	0.10278511	0.069623947	0.039448977	0.035054922	0.00430584	0.004664898
2	0.070331097	0.071228027	0.003484964	0.038974047	0.003270864	0.004153013
3	0.099446058	0.09841609	0.06582284	0.039004087	0.004165888	0.00372386
4	0.099940062	0.099736929	0.069768906	0.038875103	0.004091978	0.003881216
5	0.039813995	0.099352121	0.069499969	0.039092064	0.004070044	0.003170013
6	0.069890976	0.100258827	0.069624901	0.038988113	0.004030943	0.004392862
7	0.070018053	0.069365025	0.039613962	0.039005995	0.004014015	0.004397154
8	0.03973794	0.069938898	0.039801836	0.039009809	0.004034996	0.003859043
9	0.034041882	0.069694996	0.003488064	0.003335953	0.004040003	0.004033089
10	0.096149921	0.033917904	0.065699816	0.034610987	0.004082918	0.004536152
11	0.099498034	0.065695047	0.0696311	0.039103985	0.003813982	0.003904104
12	0.099849939	0.10026598	0.069791079	0.003330946	0.004040003	0.004210949
13	0.069996834	0.069359064	0.069508076	0.034695864	0.004028082	0.004236937
14	0.049779177	0.099880934	0.069614172	0.039014101	0.004029036	0.004143
15	0.099843025	0.079758883	0.039669037	0.003334999	0.003643036	0.003968954
16	0.06794095	0.099900961	0.003491879	0.034675121	0.004034996	0.004012108
17	0.002577066	0.099697113	0.065859079	0.038953066	0.003113985	0.004140854
18	0.069594145	0.069805145	0.03963089	0.003231049	0.004021168	0.003963947
19	0.002990007	0.033792019	0.039625883	0.034854889	0.003689051	0.003678083
20	0.096901894	0.096262932	0.039643049	0.003186941	0.004007101	0.004445076
21	0.09040308	0.06936717	0.003262043	0.034844875	0.004003048	0.004640818
22	0.099215984	0.039815187	0.066157818	0.038962126	0.004034042	0.004565001
23	0.099939108	0.070269108	0.037755966	0.039004087	0.004302979	0.004540205
24	0.099407196	0.0375669	0.071440935	0.033571005	0.003838062	0.003786087
25	0.099906921	0.101622105	0.039632797	0.034384966	0.004076958	0.003834009
26	0.09987092	0.069844007	0.039784908	0.039079189	0.00407815	0.00384903
27	0.099901915	0.09978199	0.039489031	0.038994074	0.004050016	0.00436902
28	0.069833994	0.099843025	0.039635897	0.039017916	0.003705025	0.00404191
29	0.101825953	0.101678133	0.040146112	0.03910017	0.004173994	0.004628897
	Averages					
	0.07632074	0.07734218	0.04740233	0.03150535	0.00466444	0.00425575

Table 1 and Table 2 show the time to completion for each algorithm and message size combination for 30 trials each, as well as the average time taken for message sizes in each algorithm. In all cases, with this particular cluster configuration and this set of problem sizes, lower latency was achieved when using the variably parallel program. The impact of the extra time to completion could be unacceptable for latency-sensitive applications. In addition, the recorded times for the variably parallel algorithm had a lower variance than those of the other implementation. Less consistency in performance could potentially be another detriment to the uniformly parallel FFT.

The one-dimensional FFT can be performed in various ways with a differing impact on the system in terms of load balancing, total latency, and network usage. For distributed-memory systems in which it is desired to prevent some nodes from being overused (greater load balancing) it may be beneficial to perform FFTs in parallel such that the degree of parallelization remains uniform, as presented in the second algorithm. However, this can come at the cost of higher overall latency. Even though this approach eliminates having the majority of data operations occurring on the root process, it still requires that the root process participates in synchronous communication. By waiting for data like this, the root process may end up taking more time to complete than if it would have required to simply perform data operations. This data may not be representative of the differences in latency that one could expect in more expensive computer clusters.

Conclusions

We have seen that it is feasible for an FFT algorithm to be parallelized for a distributed memory system such that the number of active processes participating in the algorithm remains uniform through its entire duration. However, the method presented here was slower in these cases for this hardware. This could be due to the extra communication overhead, as Meiyappam [3] predicts.

The choice to implement this alternative algorithm depends on context. It is possible that, with less contention for a small number of processors in later stages of the algorithm, this procedure might be useful for more efficiently balance the cluster’s load. The significance of the decreased latency can be determined on a per-case basis. Further optimization can be performed on the implementation of the uniformly parallel FFT implementation, and more quantitative investigations into its performance are a clear next step.

References

[1] S. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*, San Diego: California Technical Publishing, 1997.

[2] A. Oppenheim and R. Schafer, *Digital Signal Processing*, Connaught Circus: Prentice-Hall of India Private Limited, 2002.

[3] S. Meiyappam, "Implementation and Performance Analysis of Parallel FFT Algorithms," *ResearchGate,* School of Computing, National University of Singapore, Singapore. [Online]. Available: https://www.researchgate.net/publication/228991757_Implementation_and_performance_evaluation_of_parallel_FFT_algorithms. [Accessed: Jan. 28, 2019].

[4] S. Byna, W. Gropp, X. Sun and R. Thakur, "Improving the performance of MPI derived datatypes by optimizing memory-access cost," *2003 Proceedings IEEE International Conference on Cluster Computing*, pp. 412-419, Hong Kong, China, 2003, doi: 10.1109/CLUSTR.2003.1253341. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1253341&isnumber=28041. [Accessed: Feb 1, 2019].