Parallel Fast Fourier Transform implementations in Julia

Members- Tanay Sojwal 15BCE0001
Shivam Yogi 15BCE0201
Vaibhav Shah 15BCE0266
Lakshit Lunawat 15BCE0121

Faculty- Prof. Manoov R
Slot- D2+TD2

Abstract

This paper examines the parallel computation models of Julia through several different multiprocessor FFT implementations of 1D input. Minimizing communication overhead with the use of distributed arrays is at the heart of this study, with computational optimization details left aside as a less important factor. Throughout the discussion, input problem size and the number of processors are both assumed to be powers of 2 for the sake of simplicity. Various methods of transmitting data within Julia to reduce latency cost are considered.

Preliminaries

Before diving into the discussion of implementation details, a high-level introduction to FFT and parallel FFT methods is deemed necessary:

A *Discrete Fourier Transform*, or *DFT*, of a sequence $x = [x_0, x_1, \dots, x_{n-1}]^T$ is a sequence $y = [y_0, y_1, \dots, y_{n-1}]^T$ given by:

$$y_m = \sum_{k=0}^{n-1} x_k \omega_n^{mk} \quad or \quad y_m = \sum_{k=0}^{\frac{n}{2}-1} x_{2k} \omega_{\frac{n}{2}}^{mk} + \omega_n^m \sum_{k=0}^{\frac{n}{2}-1} x_{2k+1} \omega_{\frac{n}{2}}^{mk}, \qquad m = 0,1,\dots,n-1$$

where $\omega_n = e^{-2\pi i/n}$

This is immediately recognized as a combination of two smaller problems of size $\frac{n}{2}$, with the former containing even-indexed elements of the original array and the latter containing odd-indexed elements. Defining $y_k = x_{2k}$ and $z_k = x_{2k+1}$ yields the following two sub-problems:

$$Y_m = \sum_{k=0}^{\frac{n}{2}-1} y_k \omega_{\frac{n}{2}}^{mk} \quad and \quad Z_m = \sum_{k=0}^{\frac{n}{2}-1} z_k \omega_{\frac{n}{2}}^{mk}, \qquad m = 0,1,\dots,\frac{n}{2}-1$$

Once these problems are solved, the solution to the original problem can be determined by:

$$X_m = Y_m + \omega_n^m Z_m, \qquad m = 0,1,\dots n-1$$

However, since $\omega_n^{n/2+m} = -\omega_n^m$ and $\omega_{n/2}^{n/2} = 1$, the last $\frac{n}{2}$ terms can be reduced to (with some factoring details that shall be omitted): $X_{m+n/2} = Y_m - \omega_n^m Z_m, \qquad m = 0,1,\dots,\frac{n}{2}-1$

These reduced computations are depicted below by the famous Cooley-Tukey butterfly:

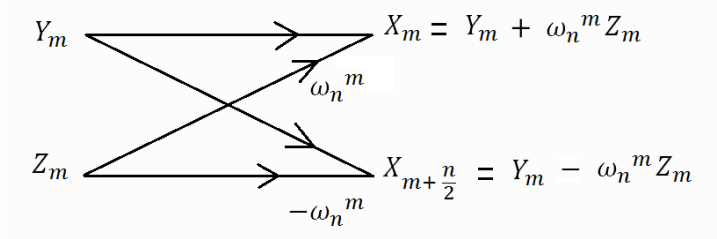

Figure 1 - Cooley-Tukey Butterfly

Note that the ordering of output elements is in a bit-reversed order, (i.e. input element at index 011 will correspond to output element at index 110 of the resulting array). Due to the recursive nature of the solution, the following pseudo code that outlines the main stages of computations follows trivially (with optional bit-reversal step depending on the desired output type):

```
FFT( array )
        … base case handling …
        FFT( even set of array )
        FFT( odd set of array )
        Combine results using Cooley-Tukey butterfly
End
```

It's evident that in computing FFT, data elements are exchanged very frequently in order to compute the butterfly relation. Because of this, parallel computing models (except for shared memory systems) must take care of handling communication across multiple processors carefully since latency and bandwidth cost are often the bottlenecks. In the development of parallel FFT algorithms, perhaps the two most dominant ones are the Binary Exchange algorithm and the Parallel Transpose algorithm. The major difference between these two methods is the approach to handling communication between different nodes. In Binary Exchange, data is distributed evenly among p processors and only the first $\log p$ stages of the computation require data exchange, with the remaining stages doing local computations.

Figure 2 depicts an input of size 8 being distributed among 4 processors; with the first 2 stages require communication, whereas the last two can be done locally since the data needed are already available on-site. This algorithm is best on hypercube network (1) since the data required will always be found in adjacent nodes, thus minimizing the cost of messaging.

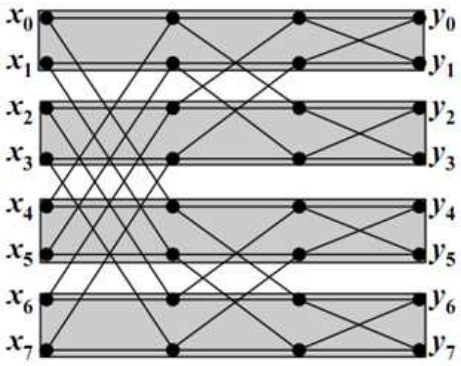

Figure 2 - Binary Exchange Example (courtesy of (3))

The Parallel Transpose Algorithm is another attempt at solving the internode communication problem. An input of size n is conceptually represented as a $\sqrt{n}$ x $\sqrt{n}$ matrix wherein only one phase of communication is needed in between computations. The following figure depicts how this actually works:

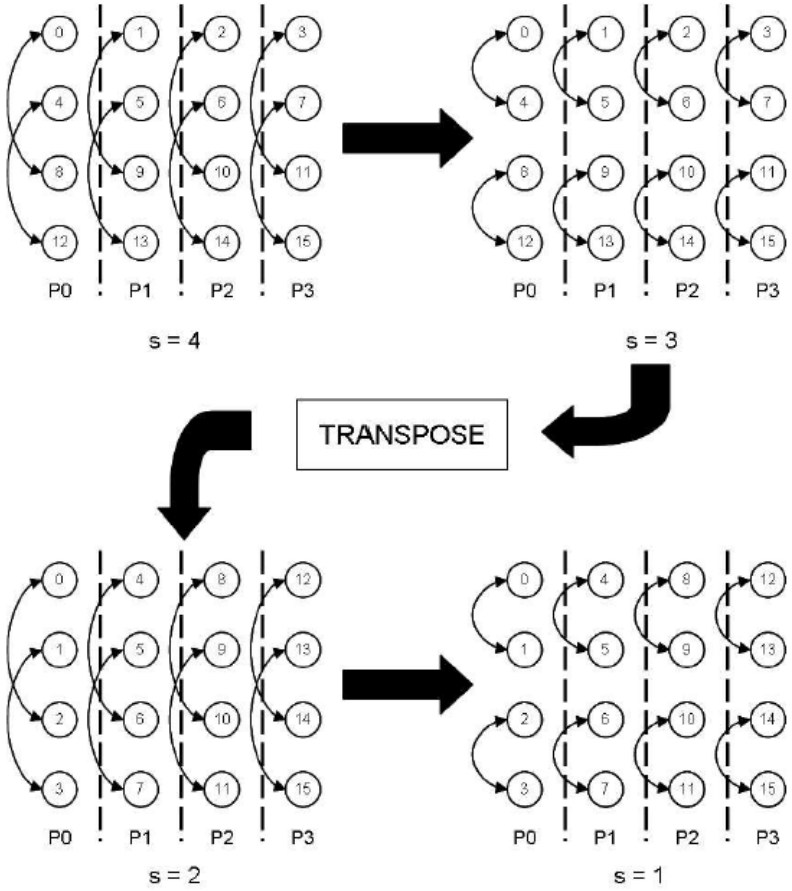

Figure 3 - Parallel Transpose Algorithm (courtesy of (2))

Here we have an array of size 16 being distributed across 4 processors, with the appropriate elements living on the bottom-labeled processors. In the first two stages, all butterflies are done locally; afterwards data are exchanged altogether in one phase to end up at the second configuration that allows for local calculations until the end results. In contrast to Binary Exchange where communication is invoked as an on-demand request between corresponding nodes, latency cost is only incurred once here. This could help reduce significant overhead in systems where communication initialization costs are expensive.

Binary Exchange and Parallel Transpose are simply different ways to tackle the communication cost. However, it is important to note that with different systems and programming languages, variations of them could be tailored specifically for the underlying architecture to improve performance. For what follows, a study of methods for performing data exchange using Julia on clusters is presented.

Julia Implementations

Perhaps the best way to represent input FFT data in Julia is with distributed arrays, or DArray objects. These are simply arrays with elements living on a subset of available processors. Instead of manually having to set up data, DArray has a nice built-in support for all array operations which also abstracts away the underlying work for cross-node communication.

With this in mind, a very straightforward yet naive way to parallelize FFT is to utilize the *@spawn* macro in Julia and modify the recursive sequential version as below:

```
FFT( array::DArray )
        … base case handling …
        @spawn FFT( even set of array )
        @spawn FFT( odd set of array )
        Combine results using Cooley-Tukey butterfly
End
```

This code will work, and in fact is the same approach that FFTW takes when implementing on shared memory systems using CILK; however, its performance is far from practical for this case. In distributed-memory clusters, the random spawning of FFT calls on processors cause way too much communication overhead since they may be assigned to calculations involving data that are non-local. This coupled with the fact that single DArray element access is slow, completely kills execution time. One can improve this spawning process by leveraging the *@spawnat* macro to specify which processor should handle the problem:

```
FFT( array::DArray )
        … base case handling …
        @spawnat owner(even array[1]) FFT( even set of array )
        @spawnat owner(odd array[1]) FFT( odd set of array )
        Combine results using Cooley-Tukey butterfly
End
```

In the above pseudo code, tasks are assigned to processors which contain the first element of the problem array in question. With this approach, the communication time is certainly reduced, however if the array at hand is distributed on multiple processors; unnecessary non-local accesses still happen quite frequently. Yet, if the array distribution is carefully taken care of, data could be guaranteed to be local for as many stages of recursion as possible and latency cost is reduced. In fact, when the input array is evenly distributed among p processors in a bit-reversed order, this algorithm follows exactly the Binary Exchange model. In figure 4 (with processors 1-4 owning data that are separated by horizontal dashed lines, from top to bottom), the first two stages of recursion correspond to the green-boxed area in which all butterfly computations are done locally within the assigned processors. Only in the last two stages (orange and red, respectively) does it require non-local communication. As discussed previously, in general only the last $\log p$ stages incur overhead, and this algorithm is a significant improvement over the previous one. In practice, however, for large input, the performance is

still intolerable partially due to the high cost of DArray element access at the last few stages. Furthermore, recursive spawning in Julia is also very expensive and appears to have a non-linear cost growth with respect to problem size. Therefore, modifications are necessary in order to improve messaging time associated with these two issues.

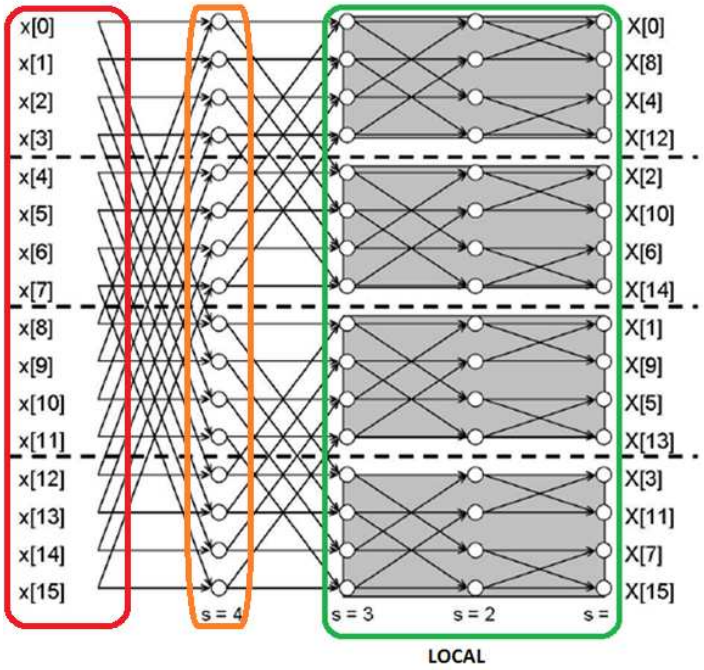

Figure 4 - Binary Exchange Implementation (courtesy of (2))

To eliminate the spawning overhead, one could treat all sub-problems in the first local stages as separate and perform a black box FFT computation on each one. For example, in figure 4, four groups of data are each solved independently and locally. Results are then combined appropriately in the last two levels. This approach resolves to the following simple pseudo code:

```
FFT_BlackBox( array )
        …Sequentially solve…
end
FFT( array::DArray )
        … base case handling …
        for each processor p
                @spawnat p FFT_BlackBox(array)
        end
        Combine results using Cooley-Tukey butterfly
End
```

Note that these black-box FFT solvers must produce unordered output, which means that the bit-reversal step must not be done when solving. The reason is to ensure a correct order of data for later stages to process. In this algorithm, any FFT solver would suffice for computing the local sub-problem given that it meets the condition stated before. Aside from reducing the number of

spawning, this also allows for the opportunity to leverage sequential black box solvers that are already optimized, like FFTW. For the purpose of this paper, a simple sequential FFT solver was written to serve as a means to illustrate the concept. In practice FFTW can be configured to handle these types of input and output.

Another remaining issue is the cost of accessing DArray elements at the last $\log p$ stages of the computation. One way to avoid this is to rearrange the data distribution to gather what is needed on a processor that is assigned to compute. In figure 4, at the orange level, data can be redistributed altogether in a bundle so that array elements that live on processor 2 are moved to processor 1, and from processor 4 to 3. The decision on which processor involved in the calculation to move data to is random here, but in practice perhaps physical node locations should be taken into account. With this approach, after the rearrangement, all data access will be local for the current stage, thus avoiding multiple distant element accesses. However, a disadvantage to this model lies in the fact that data must ultimately be gathered in one single node which might not have sufficient memory. For the purpose of this study, this consideration is left aside to focus on optimizing internode communication in the network.

With Julia, the data redistribution can be done by constructing a new DArray with elements living on the appropriate processors. Calculations are then resumed on this new array. However, even though the overhead is now significantly reduced compared to previous attempts, it still consumes a considerable amount of time, as the timing table below shows (with time measured in seconds):

Problem Size	FFTW	Communication – 4 procs	Communication – 8 procs
2^1	0.0002	0.08972	0.145179
2^{14}	0.0009	2.703539	0.191254
2^2	0.128	0.783675	1.014697
2^{25}	6.3023	21.3901	26.08323

The raw communication cost alone far exceeds actual execution time of sequential FFTW on various sizes. This is largely due to the expensive cost of rearranging data from one node to another. For an input of size 2^{25} distributed among 4 processors, the unit cost of rearranging data from one processor to another is around 3 seconds. With this approach, there are a total of $\log p$ phases of redistribution each involves an exchange of total of $\frac{n}{2}$ elements, hence the bottleneck of execution time.

Given all of these above-described characteristics of the system, perhaps the best implementation of parallel FFT in Julia would use the Transpose algorithm to impose only one phase of data redistribution and minimize as much latency cost as possible. Due to time constraint, this examination does not include this implementation, however, should still serve as a means to understand different advantages and disadvantages of the underlying architecture.

In the next section, some explanations on the written Julia code and instructions on how to run them will be presented.

Code execution

In the code package, several FFT functions are implemented along with helpers. Below is a complete set of available APIs and their descriptions:

Main entry points:

- fftp(array)
    - Takes a local array and performs parallel FFT computation using the final algorithm presented above.
- fftp_oc(array)
    - Same as fftp() except this version only emulates communication that is done. This function is useful to get an accurate measurement of internode messaging overhead.

FFT functions:

- fftp_dit_darray_fftw(array::DArray)
    - Takes a distributed array and computes FFT. This is called by fftp().
- fftp_combine(array::DArray)
    - Takes a distributed array and combine the local array elements using the butterfly rule.
- fft(array::DArray)
    - Performs in-place FFT computation on the local portion of the given distributed array
- fftp_dit_darray_fftw_oc(array::DArray)
    - Same as fftp_dit_darray_fftw() except only performs communication calls.
- fftp_combine_oc(array::DArray)
    - Same as above.
- fft_oc(array::DArray)
    - Same as above.
- fftp_dit_darray_smart_spawn(array::DArray, startIdx, endIdx)
    - FFT computation using @spawnat that was described earlier in the paper.
- fftp_dit_darray_random_spawn(array::DArray, startIdx, endIdx)
    - FFT computation using random spawning that was first introduced as a naïve implementation.
- fft_dit_in_place(array, top::Int64, bot::Int64)
    - Performs in-place FFT computation on the given array, giving unordered output.

Helpers:

- howdist(s::DArray)
    - Prints out the distribution of the given distributed array on available processors.
- redist(s::DArray, pid1, pid2)

    - Redistributes the distributed array, moving all data from processors (pid + 1) to pid2 to processor pid1.
- redistbb(s, L)
    - Redistributes data all at once assuming size of the array and number of processors are both powers of 2. Depending on the specified level L, which corresponds to the different stages depicted in figure 4; data are rearranged to ensure the computations can be done locally. This method must be called successively on results for previous level until the desired level is reached. Refer to below for instruction for how to execute.
- bitr_lookup(i, n)
    - Bit-reverse an n-bit number i using lookup table
- bitr_loop(i, n)
    - Bit-reverse an n-bit number i using standard loop and bit shifts
- bitr(array)
    - Bit-reverse an array.

The following lines of code show how some of these functions are called:

```
a = complex(rand(2^20))
result = fftp(a) # Computes parallel FFT on A

m = drand(2^20)
d = redist(m,1,3) # All array elements from processors 2 and 3 are moved to processor 1

m1 = drand(2^20)
m2 = redistbb(m1,1) # Move all data at once, if on 4 processors, elements on 2 are moved to 1, and from 4 to 3.
m3 = redistbb(m2,2) # Now m2 is distributed between processor 1 and 3, m3 will contains results all local to processor 1.
```

Sources

(1) Gupta, Anshul, and Vipin Kumar. *The Scalability of FFT on Parallel Computers*. IEEE, 1993. Print.

(2) Palmer, Joseph. *THE HYBRID ARCHITECTURE PARALLEL FAST FOURIER TRANSFORM (HAPFFT)*. Brigham Young University, 2005.

(3) Chu, Eleanor Chin-hwa, and Alan George. *Inside The FFT Black Box, Serial And Parallel Fast Fourier Transform Algorithms*. CRC, 2000.

(4) Heath, Michael. *Parallel Numerical Algorithms*. Department of Computer Science University of Illinois at Urbana-Champaign, Web. <http://www.cse.illinois.edu/courses/cs554/notes/13_fft_8up.pdf>.