

Richard Hu
Professor Edelman
6.338 Parallel Computing

Fourier Transforms and Parallel Computing

Abstract

Computer hardware is becoming cheap and cluster computing is becoming more popular. However, it is still not easy to program in a parallel environment. Thus Matlab*p was created to provide the easy, familiar interface of Matlab to users while still utilizing the power of a parallel environment. A module for Matlab*p that calculates FFT's would be valuable. This module must be faster than a serial version despite the Matlab wrapper and must also experience speed-up with the number of processors used on it.

Introduction

Computer hardware is getting cheaper by leaps and bounds every year. It is cheaper to buy several older processors than the latest and fastest one. However, the problem is that each of the older processors cannot compete with the speed of the fastest processor. The solution is to have these older processors work on the same task in parallel, that is find some method of dividing a problem into smaller sub-problems which can be solved almost independently of one another. In addition, having additional processors translates into having more space to solve larger problems that may not ordinarily fit on one processor.

The major drawback to parallel computing is the difficulty of creating a parallel program. Despite the allure of such programs as SETI, not all problems are equally easy to parallelize. Essentially solving problems in parallel fall into one of two categories: the problem is embarrassingly parallel in which case it is simple to solve or the problem is not, in which case, a specialized program has to be written for the task. Currently, there is no optimal method for taking a serial problem and abstracting it into a parallel environment without completely resolving it.

One proposed solution to this problem is Matlab*p. Matlab*p is a parallel version of the popular scientific and engineering program, Matlab from Mathworks. Matlab*p appears to have the front-end of regular Matlab so that a user that is familiar with Matlab, a serial program, can almost instantly begin using Matlab*p, a parallel program. Matlab*p achieves this by abstracting the parallel environment away from the user. Currently Matlab*p supports a wide array of commands that run on Matlab including the ability to create, add, and multiply matrices.

One very useful function that has not been included into Matlab*p yet is the ability to compute parallel Fourier Transforms. Fourier Transforms are very useful in many fields such as for analyzing and filtering audio signals or in spectroscopy and many scientists who are familiar with Matlab as a means of computing these Fourier Transforms do not have resources to write their own parallel Fourier Transform programs. In addition, many of them would like this parallel FFT program to already be incorporated with software that they already use, namely Matlab. By including the abilities to calculate FFT's in Matlab*p, we solve many of those problems.

However, it is not sufficient to merely include a FFT module in Matlab*p. Because the program is parallel, the users will be expected some type of speed-up when using more processors to calculate the FFT. In addition, users will be expecting faster computation times when compared to the serial version of FFT and in most cases, user will be expecting comparable computation times from FFT in Matlab*p as they would from a stand-alone FFT program. Therefore, this thesis will not only be the incorporation of a Fourier Transform module into Matlab*p but also deal with optimizing and comparing it versus other approaches and programs.

Proposed Approach

Currently, the proposed approach to adding a FFT module to Matlab*p is to utilize an already existing and proven FFT library and incorporating it into the Matlab*p syntax and structure. The program that was chosen is FFTW, Fastest Fourier Transform in the West. FFTW is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of both real and complex data, and of arbitrary input size. It was developed here at MIT by Matteo Frigo and Steven Johnson and is freely available for use by the public.

While there are some other vendor-specific FFT libraries that may be faster on their respective platform, FFTW was chosen because it has consistently been one of the fastest FFT libraries available, no matter which platform it is run on. This platform-independence is important for cluster computing because a cluster is not always platform-homogenous. While the Intel FFT library may run faster on Intel's, it will not run well, if at all, on Sun's. FFTW will run exceptionally well on a variety of platforms as shown by figures one and two.

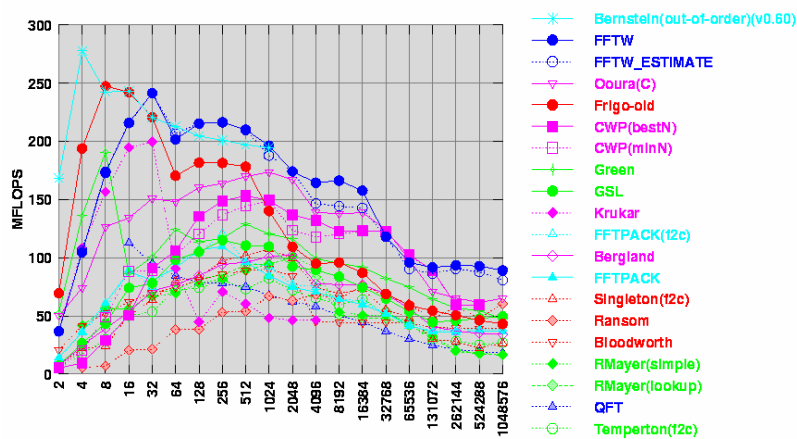


Figure 1: FFT Benchmarks on PIII 300Mhz

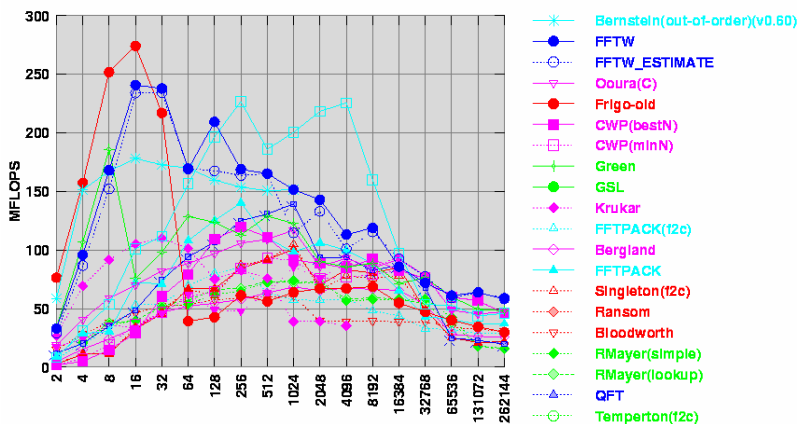


Figure 2: FFT Benchmarks on UltraSparc I 167Mhz

Data Distribution

However, adding FFTW to Matlab*p is not as simple as inserting a new function call to Matlab*p. FFTW already has a method for dealing with parallel processing but it requires the data to be in a certain format across the processors. Matlab*p, on the other hand, already stores the data in another format which is more

efficient for other tasks. The primary obstacle to integrating FFTW into Matlab*p is converting between the two data distributions.

Matlab*p utilizes block-cyclic data distribution in which data is distributed into blocks, which are placed on alternating processors as demonstrated in Figure 3.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 3 | 2 | 3 | 2 | 3 | 2 | 3 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 3 | 2 | 3 | 2 | 3 | 2 | 3 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 3 | 2 | 3 | 2 | 3 | 2 | 3 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| 2 | 3 | 2 | 3 | 2 | 3 | 2 | 3 |

Figure 3: Example of Block-cyclic Distribution

Block-cyclic distribution is more efficient because it induces load-balancing, or splitting the work evenly between processors during most computations. This property is important because by splitting the work evenly among the processors, it allows for a greater degree of speed-up as no one processor will be dragged down by an disproportionate amount of work. In addition, block-cyclic distribution is more effective because of its scalability and reduction in communication properties.

For example, in figuring out the Gaussian elimination of a matrix, using a row or column distributed matrix, where each processor contains some set of rows or columns as is the case, leads to the program being very serial as the second processor cannot begin its computation until the first processor has finish its computation. Likewise, once the second processor has finished its computation, the first processor will idle for the remainder of the computation. However, by using a block-cyclic distribution in conjunction with a recursive algorithm that operates on sub-matrices of matrices, each processor will perform an equal amount of work which leads to a greater speed-up.

While Matlab*p utilizes a block-cyclic distribution, FFTW utilizes a row-distribution that is determined at run-time. Any program calling FFTW in parallel first must call the function `fftw_mpi_create_plan` to create a plan that tells each processor which rows of the matrix that it requires. Each processor then extracts these row numbers from the plan by calling `fftw_mpi_local_sizes`. After this point, it is the responsibility of the user to put these rows on the appropriate processor in order for FFTW to work.

Therefore, in incorporating FFTW into Matlab*p, I will need to create an algorithm that translates between block-cyclic distribution and row distribution. However, because of the constraints on speed-up and time for computation, the program will need to be efficient in its use of communication as to minimize the amount of overhead time spent on sending data back and forth. In addition, the algorithm should avoid any possibility of deadlock from conflicting communication between processors.

Implementation

Translating between block-cyclic data distribution and a row or slab data distribution seems like a simple task: determine the sending and receiving processor and then issue the communication between the two processors. Unfortunately, because both the sending and receiving processor must know its destination and source respectively, this task becomes an onerous one because each and every single processor must have some method of determining whether or not it should receive or send or do neither to any position in the distributed matrix.

Naïve Approach

The naïve approach assumes that it is a simple matter to compute the desired recipient for every position in a matrix using only mathematical formulas. By using only formulas, this approach has no need for initial communication and initialization between processors and instead only transfers blocks or entire rows between processors. The only shared knowledge that each processor needs to have is the size of the entire array which will be $[M \times N]$ and the size of each individual block which will be $[m \times n]$ as well as the number of processors $[P]$. In addition, each individual block should know its context in terms of the entire matrix. For example, if the data is block-cyclic, then one block on a processor should know its relative position in the entire matrix. That information is given from a SCALAPack routine, BLACS_INFO.

From the knowledge above, given the indices of a position in the matrix, it becomes a simple matter of computing which block it belongs in. First, we treat each block as an individual position in the matrix and given them all indices as illustrated in **Figure 4**

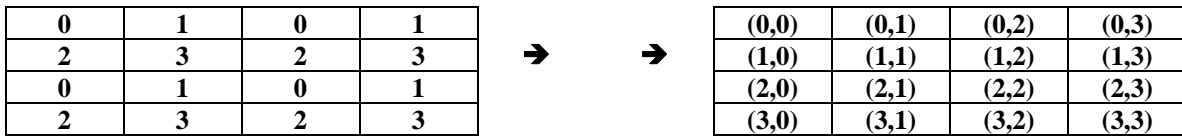


Figure 4

The row numbers are determined by the formula $\left\lfloor \frac{x}{m} \right\rfloor$ and the column numbers are determined by the formula $\left\lfloor \frac{y}{n} \right\rfloor$ where x and y are the row and column indices of that position within the entire matrix.

If we are translating from block-cyclic distributed to row distributed then we can calculate the processor that the row is supposed to be one by $\left\lfloor \frac{x}{P} \right\rfloor$. That processor becomes the receiver. The group of the sender is determined by taking the row number and finding the equivalent number modulo G , where G is the number of different rows of processors. For example, in **Figure 4**, G is 2 because there are two distinct rows of processors: 0,1 and 2,3. We then figure out the exact processor by taking the column number modulo $p[G]$ where $p[G]$ is the number of processors in group G and adding to the base processor, processor of lowest rank, of group G . Again, from **Figure 4**, $p[\text{group } 0] = p[\text{group } 1] = 2$ because there are two processors for each group or row. Translating from row distributed to block-cyclic is identical except that the sender and receiver are swapped.

The position within the block can be calculated by taking the difference between the desired index and the border of that block like as follows: $x - m * \left\lfloor \frac{x}{m} \right\rfloor$ and $y - n * \left\lfloor \frac{y}{n} \right\rfloor$. In addition, it is simple to determine which block on because each processor provides the context of each of their blocks, as in the relative position of the block to the rest of the matrix. Merely recursing through a processor's blocks can quickly determine which one is the correct one.

Precise Approach

While the naïve approach is more communication efficient and seems to work, the problem with it is that it may not correctly handle border cases when the matrices and indices are not picked well. For example, if the matrix is not evenly divisible by the block size then calculating where each position of the matrix is becomes the naïve approach with case statements thrown in everywhere. These sloppy computations can

quickly lead to unnecessary complications. The simple solution would be to be very careful initially at the cost of the slightly extra computation.

At the beginning of the program, each processor sends out a list of its borders on each of its blocks. This list is distributed with the MPI_ALLGATHER function which merely utilizes MPI_SEND and MPI_RECV. Once every processor has received information about every other processor then they can finally assemble a list of the upper bound of rows and columns for blocks. One such list is shown in

Figure 5:

| | | | | | | | | | |
|---------------|----|----|----|----|----|-----|-----|-----|---|
| Row | 9 | 17 | 25 | 33 | 41 | ... | ... | ... | M |
| Column | 11 | 21 | 31 | 41 | 51 | ... | ... | ... | N |

Figure 5

In this example, processor 0 has a block that goes from position (0,0) to position (8,10) because each number in the vectors is the upper limit of blocks.

In order to calculate which processor has which data, it is merely a matter of recursing through the lists and finding the positions of the column and row indices. This procedure will determine the row number and column number of the appropriate block. Then the precise approach follows a similar path that the naïve approach takes: it calculates the position within the block by taking the difference between the index and the border of the block which are again taken from the row and column vectors.

In addition, FFTW does not actually row distribute its data evenly. Instead, it attempts to devote a power of 2 rows to each processor in order to aid itself in calculating the Fourier transform. Barring that, it has some other scheme to throw rows onto processors. The only essential feature of the slab distribution of FFTW is that it distributes rows contiguously among processors. In other words, processor 0 gets the first X rows while processor 1 get the next Y rows and etc. Therefore the naïve approach would need some kind of row to row distribution translator. The precise approach instead uses MPI_ALLGATHER again to gather the high row numbers for each processor from the FFTW plan and again calculates which processor the data should be sent to for row distribution by recursing through the list and figuring out the position.

Results

The precise approach worked for translating from between row-distributed and block-cyclic distributed and back. I was unable to measure my actual program for time for some odd reason. However, I constructed another program to just measure the speed of sends and receives in both. I measured send and receives for a single number, a block of 100 and a block of 200. The results are in **Figure 6**

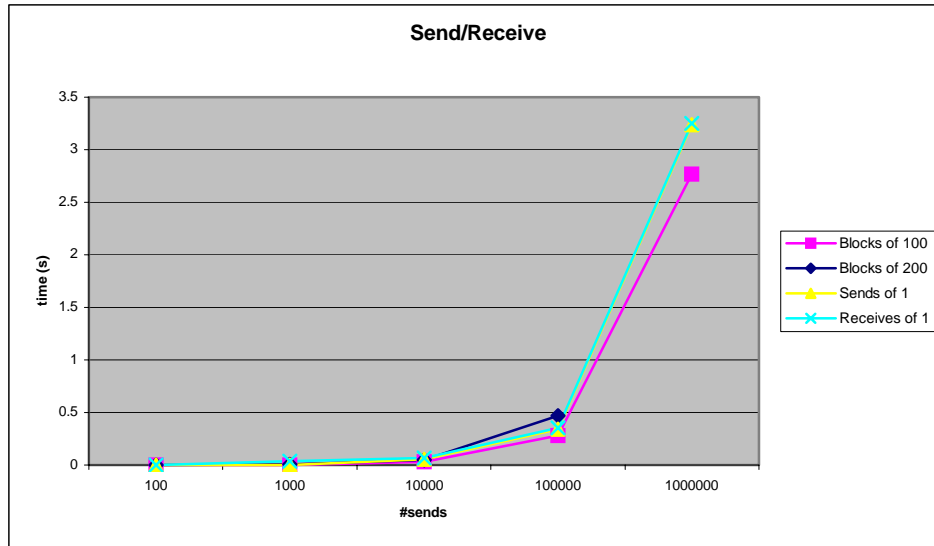


Figure 6

As can be seen from the graph, sends and receives were took approximately the same amount of time. Sending blocks was actually slightly faster than sending individual pieces of data. This probably occurs because of the overhead in latency involved in sending 1 piece of data dominates the actual send.

Some interesting behavior occurred while I was running this program that is not indicated in the chart. It seems that MPI_Send and MPI_Recv may have some sort of limit as to what they can send. Any attempt at sending blocks of 1000 or greater elements crashed the program. In addition, the attempt to send a block of 200 elements 1,000,000 times crashed the program as well. Furthermore, MPI_Send would not allow itself to be timed although it was apparent that the operation finished because all the MPI_Recv's resolved correctly.

In addition, I also attempted to count the number of sends required to convert from row to block and back. I, empirically as well as theoretically, arrived to an answer of $\theta(MN)$ where M and N are the dimensions of the entire matrix. Therefore for a matrix of approximately 1024 by 1024, the time to do all the sends and receives for row to block and back in blocks of 100 would be 0.05 seconds. Of course, if we needed a 10,000 by 10,000 matrix it would take around 6 seconds. This number is linear with respect to the number of elements. Increasing the number of processors does not help this numbers because the number of communications stays approximately the same.

I was unable to time FFTW runs because of problem compiling FFTW code on Beowulf recently. I think it may have to do with a version change or a moved library because I was able to compile the code before. FFTW publishes some of their benchmarks but they were only available in flops and they were only available for single processor machines.

Conclusion

Judging from the number of sends and receives that are necessary to convert from block-cyclic distribution to the slab distribution used by FFTW, it seems the conversion is somewhat expensive. While at 1024 by 1024, the user will not notice any difference in the speed, this may not be true at a million by a million. However, because of a lack of benchmarks of FFTW by itself across multiple processors, this communication could be negligible compared to the computation as well as the internal communication of FFTW.

In the case the communication to set up the data before the computation is significant, it may be better to consider other approaches to the problem. It may very well be that FFTW is not well-suited to a parallel environment. While the algorithm is optimized for single processors, the data distribution is non-standard which can lead to compatibility issues with other programs. Perhaps a better solution would be to design a separate parallel algorithm. Because doing a Fourier transform is nothing more than a matrix-matrix multiplication, it could be possible to create a recursive algorithm that is similar to the one used for Gaussian elimination in order to have better load-balancing and a faster compute time.

In addition, another conclusion that came about while working on this problem is that it is difficult to benchmark programs that are in parallel. There are no standards for computing benchmarks and attempting to time different segments of code can be difficult because of the parallelized nature of the program. For example, just because one section of code took 5 seconds and another section took 5 seconds does not mean both sections took a cumulative 10 seconds. For different processors, these could have different times depending on dependencies and load-balancing. Ideally, there should be some simple way of determining the following characteristics of a program: overall time, load-balancing, and communication time.

References

[1] Description of Block-Cyclic Distribution. <http://www.netlib.org/scalapack/slug/node75.html>

[2] Description of Block-Cyclic Distribution. <http://www.netlib.org/scalapack/slug/node110.html>

[3] What is MPI? <http://www.hpc.msstate.edu/>

[4] MPI Homepage. <http://www-unix.mcs.anl.gov/mpi/>

[5] FFTW Homepage. <http://www.fftw.org>