FreqDomain:  
Creates a timefreq object and outputs float array ssigned to the pixel array global object.

Timefreq  
timefreq

* Excluding stft Initial ~20.2896/21.2465/24.1805/24.1805
* Initial testing shows no dependencies in the three loops
* Trying to move the nearest calc up & its relative for loop, then attach if block
  + This doesn’t provide any speedup. Likely because C# uses a thread pool and the extra workload from unnecessary IF statements isn’t worth.
  + Seq ~20.0757/20.8147/24.5247
  + Par ~10.8159/8.6482/13.8985
* Changing the assigning into atomic yields
  + 2081.6331 -> 2100…
  + Using atomics is dumb unless you have multiple read writes to an object that is being changed. Remember that they are basically super quick mutexes.
  + They have a cost as a result
* End ~15.7647/8.1828/8.81/8.8831
* Cautious of timeFreqData, should never be in a loop as it is global.

Stft

* Initial ~2044.4187/2090.4088
* Basic parellization on the first for loop
* Attempting basic parellization on #2 2d resulted in mismatch
  + Narrowed it down to the first parallel loop not performing as it should
  + So since tempfft is global, its being overwritten by race conditions. One solution is to lockup that first portion of code.
  + Might need to use atomics for assigning how the read later on might be negatively effected?
  + Can also generate a new version every loop but doesn’t sound efficient at all…
  + Moved initialization of temp and tempFFT into the loop
    - An option was to use parallel for with thread local variables, <hasn’t been tested, will it accept two variables?
    - Can do multiple using a tuple
      * https://stackoverflow.com/questions/14831255/how-does-local-initialization-with-parallel-foreach-work
  + ~initial –
  + ~parallel – 813.8014,811.8305
  + Its far more efficient not to parallelise the inner loops
  + Ensure safety of single use globals like fftMax with an atomic op
    - Using a normal lock will increase workload theoretically. Realistically results are very similar likely because
    - Option to convert float to long to allows for interlocked read but was decided it wasn’t worth as longs are 64 as compared to 32.
* Parallizing 2nd loop with a local var of fftMax then assigning
  + ~837.1631, 837.542 > inherently slower because of the requirement of using another if statement.
  + Using a lock instead of an atomic operation saw minor speed improvements.
  + 824.7784
* Basic parallizetion on #3 2d
  + Initial ~2017.0723/2103.3248/2039.7419
  + Par ~1991.5892/2078.2113/2056.3111
  + Both loops are parallized and show the best result. This is because the work contained exceeds the overhead of generating/grabbing threads.

Fft

* Initial ~806.3612
* Parallising the inner loops raises the overall times. Granularity
* Try using task.run or parallel.invoke for the recursive functions
* Tried doing parallel.invoke with the outer loop parallelised returns significantly worse results. Performed on recursive.
* Using parallel invoke on recursive calls
  + Operating without a max depth sees a decrease in performance (assuming it iters up to 22) this is reflect in the maxdepth. Overhead exceeds operation
  + Performance increase caps at 11 as a maximum depth and has worse performance the lower this depth goes. 11 is likely ideal because it is halfway between each one so doing a 50/50 split between parallal and serial provides best perform.
  + 2500
* Parallising other loops with parallel.invoke on mD 11
  + Y = twiddles etc. – 4703
  + Even odd splitter – 4777

StftV2

* Parallelise large loop with parallel.invoke inside
  + Atomic: 2121
  + Lock: 2206/2265
* Parallelise large loop with parallel invoke and local variables –
  + Atomic : 2098
  + Lock : 2140
* Parallise first smaller loop
  + 2277
* Parallise second smaller loop
  + 2244
* Both small loops
  + 2231
* Look at without invoke
  + Second small – 2050
  + Both small – 2131
  + Outer –
    - Atomic – 831/819
    - Lock – 850
  + Parallelise first constructor one (outside of outer)
    - Atomic - 826/910 (either balanced or inneficient)
  + Parallise last setup (outside of out)
    - Atomic – 810/815 (minor improvement but should scale.)

Timefreqv

* Serial with above: 824
  + Upper
    - 829
  + Middle ~Brings almost nothing, task creation too expensive for result(is a large num of loops)
    - 825/823
  + Bottom ~big oof
    - 934

Wave file

Wavefile

* Has quite a lot of loops as subchunk rought 2382848 on jupyter
* Basic parallel loop sees
  + ~Seq 10.6
  + ~Par 3.4215

Note that I may be targeting locality indirectly when bunching loops which is a good thing

NoteGraph

Notegraph: cant be parallelised

SetRectHeights:

* Closely aligns with loadGraph function in main. Should be a simple parellisation
* Seq~1.6881/1.6682/1.6685
* Part~1.8807/1.9211
* Parellising this simply isn’t efficient.
* Tested using a smaller partition with lower granularity and looping within that block.
* If load increased by over 0.5x then it could be justified.

musicNote

no opportunities for parallelism.

False sharing!!! <https://stackoverflow.com/questions/12516084/can-parallel-for-be-optimized-for-very-short-running-operations>

Main

Load histogram

* Removed the windows forms code to make it easier to test
* Since these drawings need to occur on the same thread as the UI element, cant parallellize as receive an index out of bounds. Do more research on this.

freqDomain

* Ignore the timefreq construct
* Seq~14
* Par~4.9

onSetDetection

* Everythin under line 500 is worthless
* This func uses the custom/duplicate fft so ensure that its not parallelised and parallelised.

Fft

* Again, this is just a duplicate. Ensure to move changes over

readXML

* An IO function that has a while and for loop
* Dunno how parallising this one will go.

stringMatch (Can overload with an array)

* Many (what I assume to be small) loops, could benefit from task spawning but who knows.
* Could also squash some tasks to boost the sequential

OnSetDetection

* HFC init array
* HFC start and end of note
* If statement adds this generated period of time to understand start and finish
* Generate longer term fourier transform with more samples.
* StringMatch shows a bit of complexity and might be worth looking at.

OnSetDetection Parallisation (seq – 2158, without mainwindow stuff)

* First loop (2.6% CPU from profiler)
  + Inner loop first wth local var and lock (since atomic not valid with flaoat)
    - 2068
  + Outloop
    - 2046
  + Both par
    - 2058
  + Review
    - These two provide similar values with the inner loop being so good probably DUE TO close cache data.
    - Jagged arrays are fast than 2d arrays. CLR is optimized for single dimension array access
* Second loop
  + Tried to parallelize the whole thing and it gave lotsa bugs.
  + Twiddles loop easily parallelisable. 1607
  + Squashed loops for 1602/1610. Pretty unnecessary but keeps code maintainable
    - Parallelizing both loops results in inefficiencies as second loop doesn’t outway task cost. Can justify the piggy back.
  + FFT, tried using a func to find the perfect max depth however given that it is variable using aloop to find it adds way too much overhead and therefore using parallel invoke is not useful.
  + Naiive implementation of outer
    - Had to do initialization of many variables. Had to pass twiddles into fft as it is no longer an outaccessible variable
    - Equality error is due to pitches being applied in the wrong order since it is parallel.
      * Change pitches to array and assign via mm index rather than append to end of list. 1139. Increases runtime but maintains safety.
      * Use concurrent dict instead 1136
      * Will need to test this one on larger data sizes.

Options

There was an option to use the threadpool queue work item instead of a for loop, however it is recommended against as parallel.foreach provides far better intelligent parsing.  
<https://stackoverflow.com/questions/15122756/difference-between-threadpool-queueuserworkitem-and-parallel-foreach>  
<https://blogs.msdn.microsoft.com/pfxteam/2008/01/31/recursion-and-concurrency/>

Looks like the parallel for loop is a barrier in itself and wont return till all tasks are complete.   
<https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.parallel?view=netframework-4.7.2>

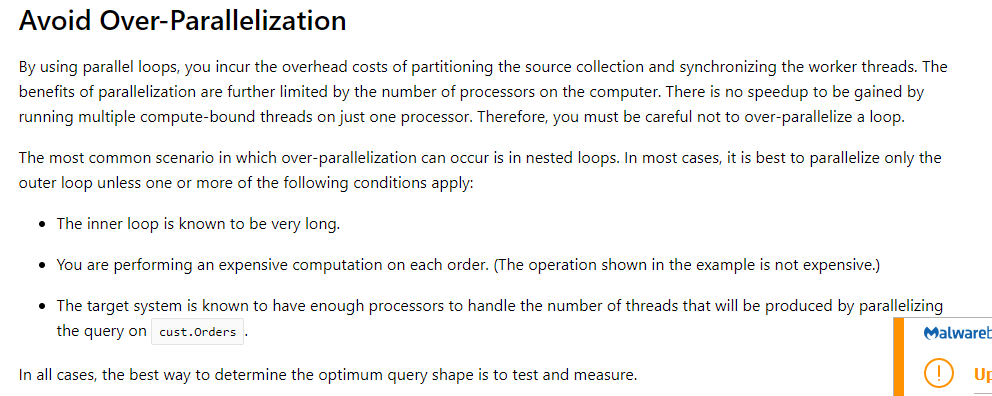
Using thread local variables instead of instansiating every time and to protect it from other thread interaction.  
<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/how-to-write-a-parallel-for-loop-with-thread-local-variables>

Gotta check if a spinlock can perform better than the lock/atomic on the max op  
<https://stackoverflow.com/questions/20800954/is-interlocked-compareexchange-really-faster-than-a-simple-lock>

Arrays are thread safe<more research>  
<https://social.msdn.microsoft.com/Forums/vstudio/en-US/7781a7fc-9964-4d3f-883d-f19935d52eb9/is-a-simple-array-thread-safe?forum=netfxbcl>

Parallel barriers  
  
<https://stackoverflow.com/questions/8756190/parallel-for-loops-are-they-wait-for-finish>

Nested loops



<https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/potential-pitfalls-in-data-and-task-parallelism>   
<https://stackoverflow.com/questions/3281604/nested-parallel-foreach-loops-on-the-same-list/3281649>   
<https://blogs.msdn.microsoft.com/pfxteam/2012/03/14/is-it-ok-to-use-nested-parallel-for-loops/>

Thread pooling  
<https://docs.microsoft.com/en-us/dotnet/standard/threading/the-managed-thread-pool>

Lock comparison (needs spinlock and tomic)

<http://www.albahari.com/threading/part2.aspx>

<http://blogs.microsoft.co.il/sasha/2013/07/12/lock-vs-mutex/>

Ui threading  
Likely not have multiple UI threads if its not multi window.

Other  
Ensure doing a release build for the final test as debug is not perfectly efficient.  
Refer to pareto principle (Lect2, 1:30)

Loop unrolling – couldn’t find anything saying it was worth it but it should only provide minor results for the amount of time required doing it and time spent trying to understand the code after unrolled.

BitSwap algorithm. Mention it is possible to further increase speeds of the algorithm but also very difficult to alter the code to suit my purposes. (If we have leftover time after reporting just try again to implement.)