MATLAB Vectorization and Parallel Computing

Kevin Kirchhoff

Abstract

Vectorization is a powerful method, implemented in MATLAB that is used to distribute computational processes among multiple threads, cores and nodes. Using parallel processing on distributed systems, such as computer clusters, it can increase algorithm efficiency and reduce computation time dramatically. This paper will explore the process of vectorization and its application to parallel computing using CPUs and GPUs.

Vectorization

Vectorization is the process used in MATLAB to optimize and simplify various operations and functions. These functions are computed as vectors, and instead of iterations, each value operation is distributed and calculated simultaneously. Like opening up new registers at the grocery store to shorten the check-out lines: when a loop is vectorized, the operations are distributed among other processors to speed up overall operation time. Vectorization is powerful and becomes even more efficient with parallel computing.

Parallel computing involves using multiple CPU or GPU cores to simultaneously preform operations in a computing intensive task. This can be anything from complicated simulations to simple sorting algorithms on large amounts of data. Vectorization is perfect for parallel processing because it involves independent computations that can be distributed, calculated and gathered to give an output. When computing loop-based operations linearly, it can take up a large amount of computing power, resulting in time consuming computations. With MATLAB and parallel processing, these tasks can be completed up to one hundred times faster.

Vectorization has many uses. Since computers work with a set system of logical rules, they cannot solve problems like humans. Computations such as trigonometric functions and integrals may be doable by hand, but on a computer it isn't as simple. They must use various numerical methods to compute functions and solve problems.

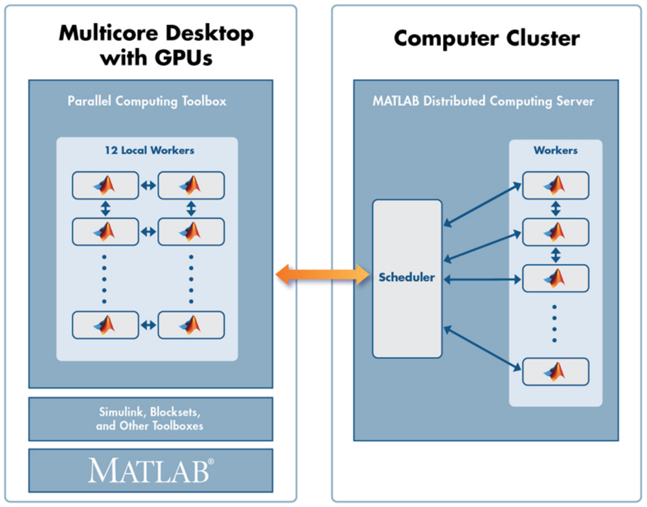
Parallel Computing

There are two methods of parallel computing: GPU computing and multi-core CPU computing. A Central Processing Unit (CPU) is the heart of a computer, used to distribute and compute the main tasks on the system. When a process, such as vector multiplication, is executed on a normal desktop computer, the CPU calculates the process linearly. In the case of two vectors being multiplied, each value of the two vectors are multiplied one by one and added to the total value until finished. It becomes obvious why this could be problematic when preforming such processes on very large vectors. This is where parallel computing and computing clusters come in handy.

Computing clusters are essentially connected computers, called nodes, which contain CPUs with single or (more commonly) multiple cores. These computers can be used to calculate large operations when distributed individually among separate cores. When calculating vector operations, individual components of the vectors can be distributed among the clusters many CPUs, which calculates the operation and sends the result back to the main CPU to be completed. This cuts computation time down significantly.

Parallel computing on large clusters is very common in many fields of research, especially fields requiring analyzation of large amounts of data. Using various nodes is convenient for distributing jobs that require large amounts of memory and complex code. Normal desktop programs require as little memory as a few bytes to as much as a couple gigabytes depending on the program. Programs designed to analyze and process data can be much more complicated and rigorous. Some programs can take up to hundreds of gigabytes of memory, resulting in incredibly intensive work for a computer.

The diagram below shows the process used in parallel computing on a cluster:


Multicore Desktop with GPUs
Parallel Computing Toolbox
12 Local Workers
Simulink, Blocksets, and Other Toolboxes
MATLAB®
Computer Cluster
MATLAB Distributed Computing Server
Workers
Scheduler

When submitting jobs on a cluster, there must be a master node that makes the commands, which is represented in the diagram in the Multicore Desktop panel. It distributes these commands to the workers, which are allocated threads on the CPU's cores.

Not all parallel processing is done on CPU clusters. A Graphical Processing Unit (GPU) is designed to compute tasks simultaneously to display computing intensive graphics. To do this, GPUs use large amounts of small, less powerful processing cores. The concepts of graphical computations are similar to those used for parallel computing. Jobs that are easier to distribute (including vector operations) are great for parallel computing on GPUs. Higher end GPUs, designed for parallel processing, can contain thousands of cores. To use GPUs for parallel computations, NVIDIA (a graphics computing company) created a processing architecture, CUDA, to be used on their products. MATLAB has CUDA methods built in that are available and easy to use.

The downside to GPU parallel computing is that programs must be written specifically for the GPU. In applications, such as simulations using 3D modeling, GPUs excel because they are designed to handle graphical processes. Applications dealing with larger programs that contain nested loops and complex algorithms simply can't be written to run on a GPU.

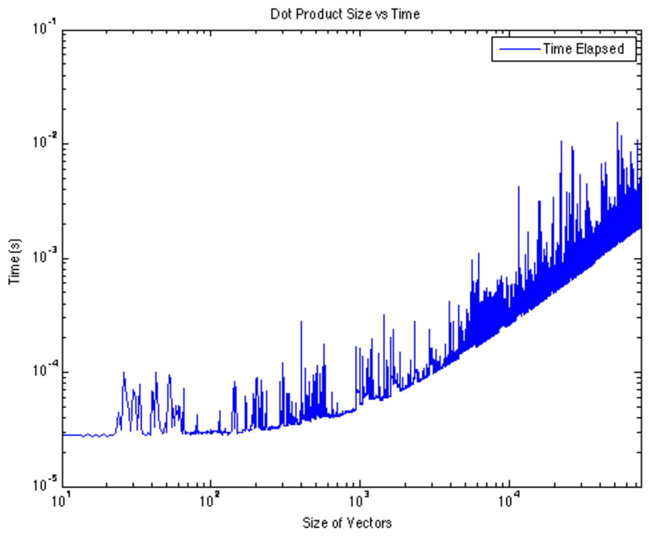
Vector Operations

Vector operations are applicable to nearly every math based scientific field. This includes physics, atmospheric science, engineering and computer science. In fact, fundemental CPU processes are based on vector operations. Real world applications require various vector operations with vectors of any size. These operations can be as simple as adding two lists of expenditures or as difficult as rendering vivid 3D surfaces with matrices.

At the base of every computing operation is the constant changing of values. Most of these values lie in matrices of any dimension. From one dimensional vectors, to incredibly large m-by-n matrices. Matrix operations play a key role in running every program. One fascinating property of matrix operations is that they are computed independently; value by value. Initially, it may not sound interesting, but when applied to concepts like vectorization, it becomes apparent how important this property can be.

When two matrices are multiplied, the resulting matrix comes from the product sum of the individual values in each matrix. Traditionally, small matrices and vectors are multiplied on a computer by simply looping through each value. This would be fine when writing code to compute a simple problem that would be tedious if done by hand. As problems become more complex and the number of values increases, these problems become difficult even for a computer to evaluate. Matrix multiplication can be broken down into a set of one dimensional vector operations, since each value is computed by multiplying the matrices one row at a time. This kind of simplification is key to what makes vectorization so important. When a problem can be broken down into sets of vector operations they become more manageable.

The plot below shows the dot product computation times of random vectors with dimensions between 1 and 75 thousand:



In the field of particle physics, correlations using dot products of vectors are essential in detecting new particles and their signals. These correlation operations are often computed over vectors containing as many as a trillion values. On a normal desktop, computing the correlation between two vectors containing one thousand points can take up to one second. With a vector of a trillion values, this operation would take weeks. But parallelizing these operations can reduce the time it takes to less than half a day.

Using MATLAB

MATLAB is a powerful tool that is widely used for parallel applications. Its built in functions make it incredibly useful. Elaborate vectorization functions and techniques are available and require only a few lines of code. The following code is a simple program to compute the sine of a thousand (and one) values in the range of 0 and 10:

```
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
end
```

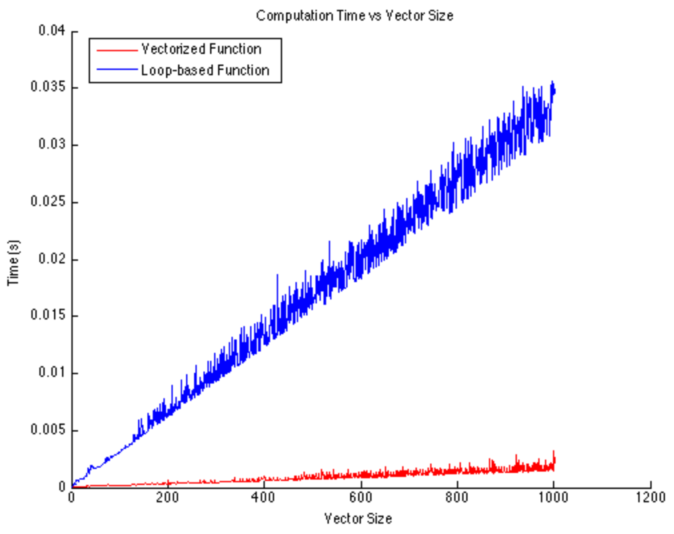
The code can be vectorized very simply in MATLAB:

```
t = 0:.01:10;
y = sin(t);
```

This algorithm is short yet powerful. When executed, the sine values are calculated independently, before being sent back and stored in y. This reduces the amount of linear iterations.

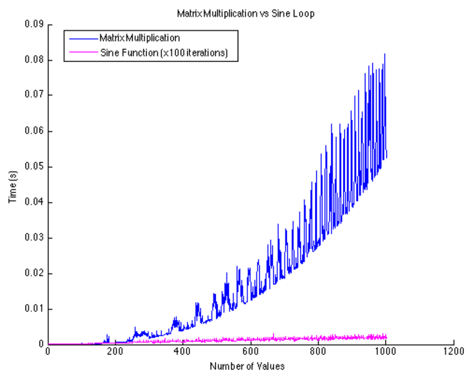
On a dual core 2.5GHz desktop CPU, the first algorithm takes, on average, ∼167 microseconds (0.000167 seconds) to finish. The vectorized version is, on average, completed in ∼67 microseconds; nearly 2.5 times faster. These times seem short, but when it comes to real word applications, one thousand operations is a very small amount: in the before mentioned physics application, as many as one trillion operations need to be processed.

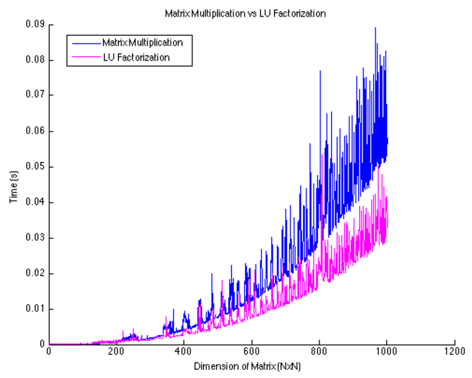
The plot below shows the increase in computing time as more components are added to each vector for both sine algorithms from above:



The non vectorized sine example (shown in blue on the plot) is computed up to 100 thousand iterations. If it were over a trillion points, the process would take more than four days; and the computation required in this calculation is trivial compared to most real world applications.

A less trivial operation, compared to the sine function, would be the multiplication and factorization of matrices. These operations on matrices in numerical methods is key to solving linear systems of equations. When multiplying a matrix by another matrix or vector, the amount of operations needed to find the solution is greatly increased. The plots below shows the amount of time it takes to multiply an n-by-n matrix compared to the sine function (first plot) and the LU factorization of the same n-by-n matrix (second plot) in MATLAB:





Both plots were computed using vectorization. The plots show how much more computing intensive some operations can be. The code for the sine function increases linearly, while square matrix multiplication increases exponentially as the number of values increase.

A real world application may be the best way to show the power of vectorization and MATLAB. In numerical methods, solutions to systems of linear equations can be computed using LU decomposition with forwards and backwards substitution, which uses vector and matrix operations. Given a system of linear equations in the form $Ax = b$, where x is the unknown, we can use matrix factorization to obtain the two triangular matrices, L and U, where $A = LU$. The original equation can be rewritten in the form

$$\begin{cases} Ly = b \\ Ux = y \end{cases}$$

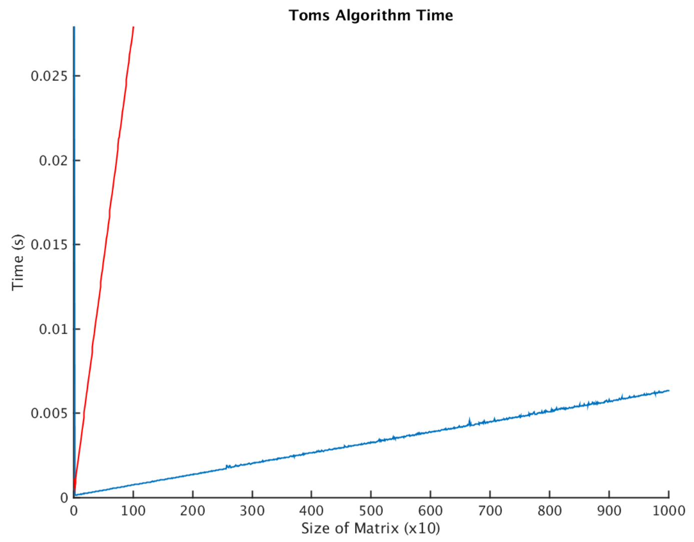
Forward substitution with $Ly = b$ can be solved for y, then $Ux = y$ with backwards substitution can be solved for x. Thus solving the system.

This method is used in the following code, which is an implementation of the Thomas Algorithm for solving tridiagonal matrices:

```
n = 100;
A = full(spdiags(rand(n,3),-1:1,n,n));
q = rand(n,1);

[L,u,P] = lu(A); %lu factorization
l = P*L; %complete lower matrix
y = zeros(n,1);
y(1) = q(1);

p = 1:n-1; %forward substitution
y(p+1) = q(p+1) - l(p+1,p)*y(p);

x = zeros(n,1);
x(n) = y(n)'/u(n,n);

r = n-1:-1:1; %backwards substitution
x(r) = (y(r)-(u(r,r+1)*x(r+1)))'/u(r,r+1);
```

This code generates a random square tridiagonal matrix of size 100 and a corresponding solution vector, q. The first part of the code factors A using LU decomposition, then uses two loops for forwards and backwards substitution, completing the algorithm.

The code uses vectorization in place of typical for loops for substitution. The nature of the algorithm makes it less than optimal for parallel computing, but MATLAB vectorization still speeds up the algorithm as seen below (vectorized code is in blue, non vectorized in red):



Even though the previous plots comparing matrix size and computing time grew exponentially for LU factorization and matrix multiplication, the Thomas Algorithm time increases linearly. This is because the algorithm is only computing looped vector operations on matrices over the individual tridiagonal areas rather the entire matrix. If the algorithm were to be using full matrix multiplication, the lines would grow exponentially.

The time difference is caused by the amount and type of operations in the program. Multiplication and division are more computing intensive than adding and subtracting. In computer science, algorithm time is measured in order notation and can be represented by the equation, $O(n)$. To compute the order of an algorithm, you must take into account the amount of loops and operations.

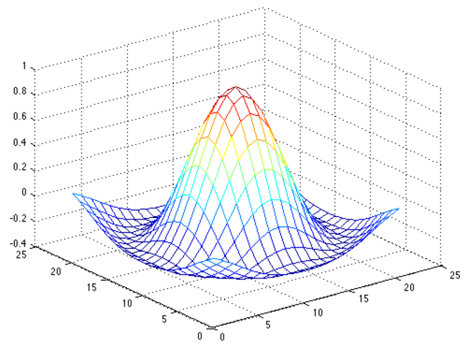
The Thomas algorithm has order n, which is much faster than any algorithm computing an operation on a square matrix as a whole. Normal square matrix operations have an order of at least n^2, since n^2 points must be evaluated. The order shows that different types of matrix and vector operations greatly affect the time it takes to complete. Avoiding unnecessary operations when developing algorithms is a key component to vectorization.

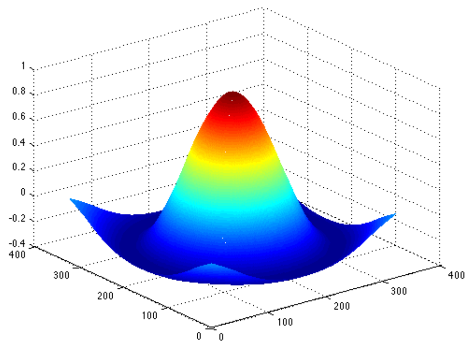
Numerically, a problem with the Thomas Algorithm, is that pivoting cannot be used, which means that it can't be iterated to increase the accuracy of the output. With matrices that are well conditioned this is not a problem. If a matrices' condition number is large, then pivoting becomes necessary. And if the algorithm were rewritten to allow pivoting, large condition numbers would increase the computing time. So the Thomas Algorithm is most commonly used with less random systems.

A simple randomly generated example doesn't properly show the purpose of solving tridiagonal matrices. One real application would be cubic spline interpolation, which often uses a variation of the Thomas Algorithm. Cubic spline interpolation is commonly used to smooth 3D surfaces. Below is an example of this in MATLAB:

```
%creates the grid and assigns the domain values to the ...
    output arrays
[X1,X2] = ndgrid((-5:1:5));
R = sqrt(X1.^2 + X2.^2)+ eps;
V = sin(R)./(R); %matrix of sample values

Vq = interpn(V,1,'cubic'); %interpolates V
mesh(Vq) %displays the plot
```

This code generates a 3D plot of Vq from the function V:

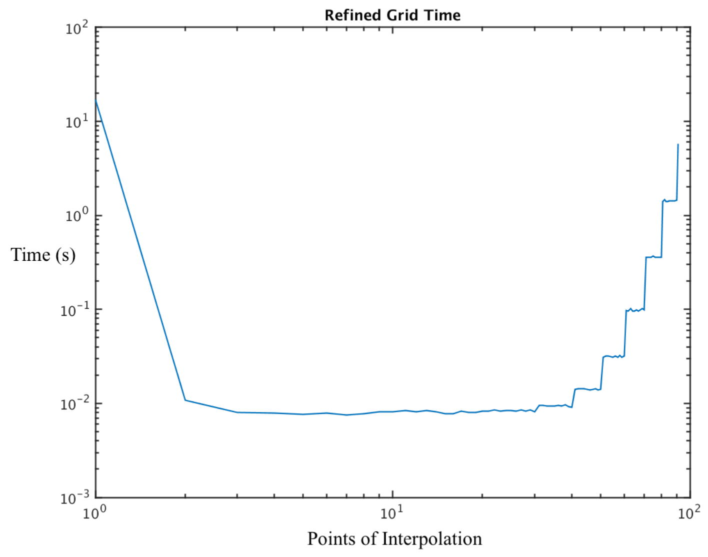




The code works by generating a discrete set of values from the domain of the grid, then computes the function at those points to give a set of sample values to be plotted. The surface is then interpolated using forwards and backwards substitution from a cubic spline, much like the Thomas algorithm.

The first plot shows the output of the code above, which has one interpolation point between sample values, in Vq, from the equation V. In the MATLAB *interpn*() function, the second parameter (lets call it k) determines the amount of interpolation points to add using the equation $2^k - 1$. The smoothness of the surface depends on the amount of interpolation points. The second plot shows interpolation points between the original sample points, Vq.

The reason for interpolating is because the actual calculation of the function takes up too much computing power. Using a system of equations with matrix operations is much faster, since it can be vectorized. When the amount of interpolated points is increased, the surface becomes more defined, but begins to slow computing time. The plot below shows how quickly the computation time slows as the points increase:



If the function were more elaborate, or if it was computed over a larger grid, this would become a serious problem. Rendering 3D surfaces for simulations, video games and other complex graphical programs require a lot of computing power.

Vectorization With Parallel Computing

In the previous section, it was shown that vectorization takes significantly less computation time, but they can still take up significant amounts of time depending on the computing power required. This is because they were computed on a desktop with only one dual core CPU. Parallel computing, used together with MATLAB and vectorization, speeds up these processes even further.

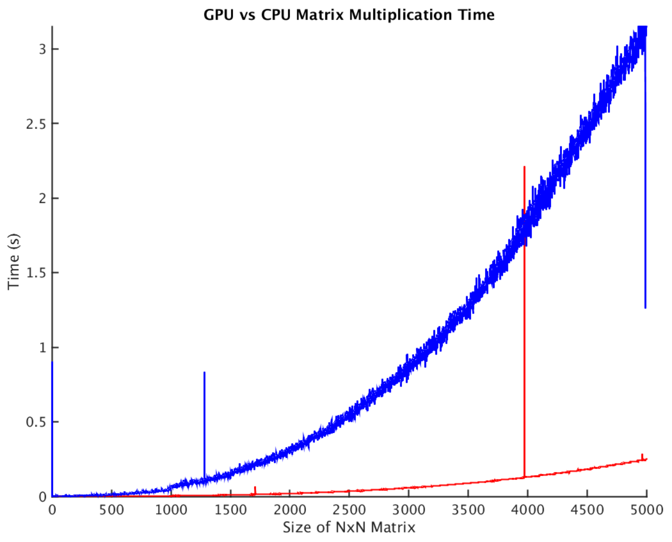
When computing vector operations, the amount of floating-point operations per second (FLOPs) determine how quickly the program will finish. In the Thomas Algorithm example, order notation was used because of the linear nature of the algorithm. When dealing with massive vectors, the notation $O(n)$ doesn't describe the computing time very well. By knowing how many FLOPs a system can perform, it is easier to determine the time it will take.

Combining the power of vectorization along with cluster based parallel processing, computing intensive programs can be computed at a reasonable rate. MATLAB automatically distributes vectorized jobs to every core available. This makes parallel jobs easier to execute, especially on a GPU. A single NVIDIA Tesla K40 GPU, which is designed for parallel jobs rather than displaying graphics, has over 2.5 thousand cores. Writing a MATLAB program for parallel computing is as simple as the code below:

```
a = ones(1,1000)*0.01;
x = gpuArray(a);
y = sin(x);
result = gather(y);
```

This code is just another way to write the sine calculation function in the previous examples. The difference is that the values of the array are contained on the GPU. Distribution occurs at the time of allocation. When the sine function is executed, it is executed using every available core which contains an evenly distributed amount of array elements. After all the values are computed, they are collected and sent back to the CPU with the gather method.

Vectorization is so effective on GPUs because they are designed to compute large amounts of simple tasks at once. GPU cores are not as powerful as CPUs, but vector operations are relatively simple. When a problem is computed using only a few vectors of larger dimensions, GPUs are great. They have massive amounts of cores, implying that they can host massive amounts of vector operations. CPU parallel computing is more suited for operations involving a larger series of smaller vectors, since CPUs can distribute these vectors more efficiently. The plot below shows the time it takes to multiply a square matrix by itself on a CPU (blue) vs a GPU (red) as the size of the matrices increase:



The lines in the plot separate very quickly. The GPU time rises at a much slower rate because of how many FLOPs it has compared to the CPU. The particular GPU that was used to generate that plot had over 10 Tera-FLOPs. This is common for most high end parallel computing GPUs. An average single core 2.5GHz CPU has only 10 Giga-FLOPs. This implies that it would take over 100 CPUs to compete with one GPU.

The only way to access 100 CPUs is through distribution over multiple computers contained on a cluster. Computing clusters are typically very large, they are most useful for problems that are easily distributed among multiple nodes or algorithms that cannot be efficiently processed on a GPU. Each node usually contains a larger amount of CPU cores compared to the average desktop. The KU ACF cluster contains over 400 nodes with as many as 64 CPU cores each. Using multiple nodes with many cores, a cluster of CPUs can be as fast and useful as a GPU.

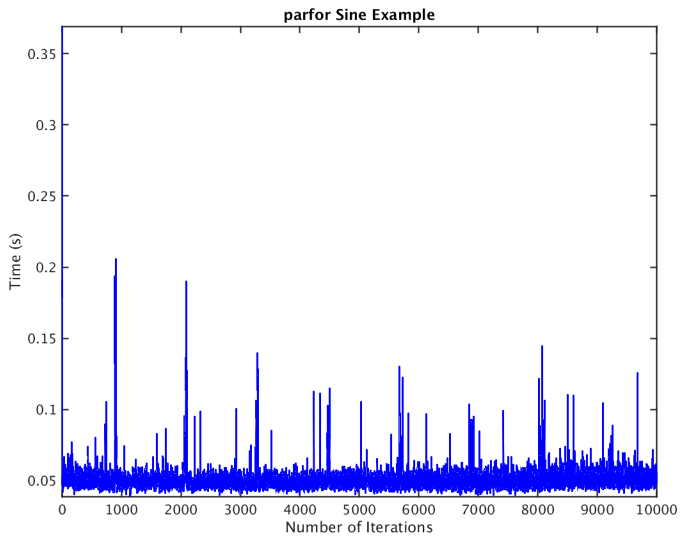
MATLAB automatically uses all of the available cores' processing power. This is very convenient for distributing programs among clusters. One common way of distributing jobs among a cluster is through batch scheduling, which is a set of operations to be performed over a selected amount of nodes, a program can be distributed over all the chosen nodes and their respective cores. Once the process is finished the results are sent back to the main CPU, where they are put together to present the output.

A more simple method is provided by MATLAB using the *parpool* operation to distribute a process within a script with less code. *parpool* is used to describe what kind of cluster the program is being executed on to determine how many workers to use. A simple use of *parpool* to optimize speed is the $parfor$ loop, which simply uses the *parpool* instructions to distribute the loop process among all the available cores. The sine function example can be converted into a $parfor$ loop like so:

```
parpool('local',16);
j = 1;
parfor t = 0:10
    y(j) = sin(t*0.01);
    j = j + 1;
end
```

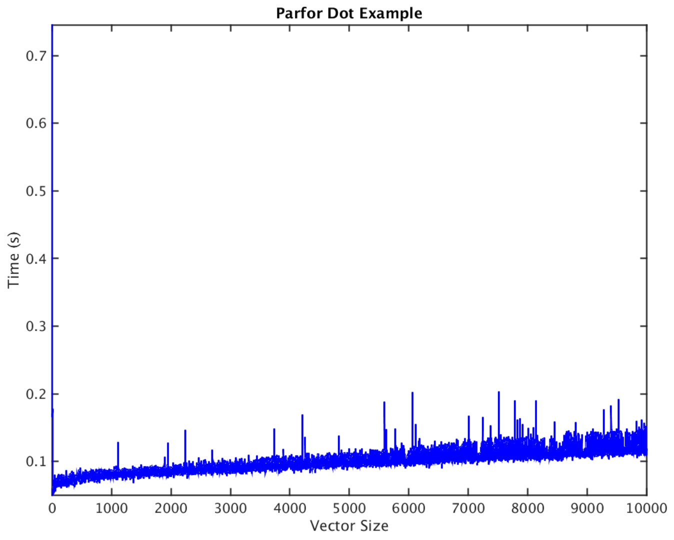
The $parfoor$ loop executes the for loop amongst the available workers in parallel. The 'local' input is the cluster profile. It describes the layout of the cluster, how many nodes are available and how many cores are on each node. The second input is the desired amount of workers. Technically there can be multiple workers on one core, but it is better to use one worker per core to avoid slowing down the program. In this example there are 16 workers set on one node (which contains 16 cores).

The plot below is the sine example implemented with 6 workers over 10 thousand loop iterations containing the $parfor$:



The spikes are caused by the precision variance in the calculation of the sine function. The average iteration time remains relatively constant, but much longer compared to the first vectorized example of the sine function. This is because the operation in the $parfor$ is a computationally simple example. The time it takes to calculate the operation is also constant as the vector size grows larger.

A better example of vectorization using the $parfor$ loop would be a simple dot product. The plot below shows the time it takes to calculate the dot product as the vectors grow larger. The parpool settings used 12 workers.



As the vectors increase in size, it takes longer to compute them, even when distributed by the $parfor$. This is because the operations are growing is size and difficulty. There are less spikes with smaller variance because the operations computed are regular floating-point operations rather than a function. This again, is where the computing time depends on the amount of FLOPs. Twelve cores provide a decent amount of FLOPs, but still not enough to compare to a GPU.

Over multiple nodes, MATLAB can distribute operations among hundreds of workers. Unfortunately, the KU ACF Cluster doesn't allow for MATLAB processing on multiple nodes, but CPU clusters could potentially preform better than on the GPU, when given enough cores.

Conclusion

MATLAB is a powerful tool. It is built to vectorize and optimize programs to their fullest extent. With powerful computers and the technology that they use, it is amazing at how efficiently operations can be processed. In today's world, computing speed is a hot topic. With new discoveries and advancements in research, more effective methods must be developed to make further advancements.

Vectorization, GPU parallel processing and CPU cluster parallel processing are all based on the same general concept: distribution of vector operations. The fundemental concept of vectors is what makes this process possible. Any problem that can be broken down into a set of scalar vector operations can be solved at an exponentially faster rate. Unlike the human mind, computers work with a strict set of logical rules. Simple calculations, such as trigonometric functions, are impossible for computers to process properly. As a result they must be computed using various approximation techniques. Numerical methods such as these are the fundamentals of computing.

References

[1] Documentation. (n.d.). Retrieved December 19, 2014, from http://www.mathworks.com/help/matlab/

[2] Shampine, L., Allen, R. (1997). *Fundamentals of numerical computing.* New York: John Wiley.

[3] Karniadakis, G. (2010, October 10). Thomas Algorithm for Tridiagonal Systems. Retrieved December 19, 2014, from http://www.cfm.brown.edu/people/gk/chap6/node13.html

[4] Parallel Programming and Computing Platform — CUDA — NVIDIA — NVIDIA. (n.d.). Retrieved December 19, 2014, from http://www.nvidia.com/object/cuda_home_new.html

[5] ACF Main Page. (n.d.). Retrieved December 19, 2014, from https://acf.ku.edu/wiki/index.php/Main_Page