Title: Biometric Fingerprint Recognition Via Multi-Core Computing
Author: Samir Faci
Adviser: Dr. Mitchell D. Theys
Secondary Reader: Dr. John T. Bell

1. *Introduction*

The purpose of this project is to demonstrate the uses and benefits of multi-core computing as it relates to a real world application, such a fingerprint recognition. A rudimentary fingerprint analysis system was developed and setup for the benefit of this exercise. Although a biometric portion was implemented, it was implemented as an ends to a mean. It is by no means a complete system nor does it strive to be. The goal of this exercise is to utilize the power of multi-core processors to see what kind of benefits can be extracted from using multiple cores. If adding more cores to a problem helps us find a solution, or if it creates more problems then it solves. It is my hope that beyond simply learning about a new technology, this project demonstrates that the uses of modern technology can indeed increase performance, but as everything one should strive for a balance. The amount of gained processing per node should offset the bandwidth cost of transferring data from one node to the next in order to accomplish said task.

2. *Hardware and limitations.*

The hardware used to test and complete the project architecture was a Solaris machine with an 8 core Niagara processor. The machine was setup to dual boot SUN Solaris and Gentoo Linux. All testing was done using the Gentoo Linux operating system.

System was compiled using gcc-4 series, standard headless Linux setup, with the openmpi libraries for parallel computing. Standard command line utilities, kernel, and basic libraries were installed. No graphical environment or any services aside from the bare necessities were installed.

Specs:

- 8 Gb of Ram
- 8 Cores, with 4 CoolThreads per core.
- 1.2 Ghz processor.
- T1, Niagara processor.

Although the Niagara[1] cpu has 8 computational cores, it does have a limitation of sharing a FPU among all 8 cores, so any floating point intensive calculation tend to not perform as well on the Niagara systems.

A secondary system was used to test and develop the early stages of development, and was also used for benchmark comparison. The secondary development machine was a Lenovo T61 laptop with an Intel Core 2 Duo[2]

The T61 is running Gentoo Linux as well.

Specs:

- Intel Core 2 Duo
- 2.2 Ghz processor
- 2 Gb of Ram

3. *Objective*

To demonstrate the uses of parallel computing as it relates to a real world application. This program implements a rudimentary biometric fingerprint recognition system that would identify a unique image when compared to a collection of sample images using anywhere from 2 to n cores. (though only tested for up to 32 cores)

4. *Biometric Aspect*

Several test images from NeuroTech[3] and East Shore Technology[4] were used to test and fine tune the algorithm used for both the image processing and the data analysis. Standard practice for fingerprint identification involves binarizing an image (turning the image into a black and white image, rather then a grayscale image) and thinning the image, where each line is a single pixel wide to ease the pattern matching portion.

Binarization: Initially OTSU's algorithm[5] was used to find a focal point that will be used to turn any value greater then the pixel value returned by OTSU to black and less then to white. Unfortunately OTSU's algorithm uses far too many floating point calculations which couldn't be overcome. An arbitrary value was chosen as a midpoint rather then relaying on a binarization algorithm due to the hardware limitations.

OTSU's Method[6] works by traversing the entire image and making a histogram of all the values found in the image. In our case, all possible values of the pixel would range from 0 to 255. OTSU traverses the image and counts the number of occurrences of each value. It finds the threshold and minimizes the weight within-class variance

[4]OTSU's Assumptions: Histogram is bimodal
No use of any spacial coherence
Assumes stationary statistics.
Assumes uniform illumination

OTSU behaves very badly when certain parts of the image have particularly darker areas. If the image brightness isn't relatively consistent the image will turn out either overly dark or overly bright and missing key fragments of the image. OTSU Algorithm when tested with our sample data worked out relatively well, the only issue being the hardware limitations which forced us not to rely on OTSU for binarization.

Thinning: For the thinning process Bowditch's Algorithm[7] was used to thin down the image down to a single pixel. Hilditch worked reasonably well, aside from the issue of white spots inside the image that are interpreted as part of the image rather then being ignored or removed from the final image.

Hiditch Algorithm works analyzing each pixel in the image and looking at all its neighbors. It uses a set of rules to determine if each pixel needs to be pruned or not.

Depending on the value of its neighbors it tests for 4 possible conditions. If all 4 conditions are properly met, then it will proceed to nullify the pixel. How the pixel is nullified depends on the developer's implementation, in this case the a black pixel is simply turned white.

Hilditch's Conditions:

I'll refer to the pixel being look at as P1.
B(P1) is the number of black pixels neighboring P1.
A(P1) number of 0,1 pattern in the sequence P2,P3,P4,P5,P6,P7,P8,P9,P2

Condition1: 2< =B(P1) <=6 , black neighbors of P1

The number of non-zero neighbors of P1 is greater or equal to 2 and the number of non-zero neighbors of P1 is less or equal to 6.

Condition2: A(P1) = 1

Ensure that the pixel is connected. That is by pruning p1, we won't create a break in the image which may lead to false positive in certain scenarios. Such breaks in a line may lead to to a false positives in fingerprint analysis, since they are key features being looked at to help establish the uniqueness of a person's fingerprint..

P9	P2	P3
P8	P1	P4
P7	P6	P5

Condition3: P2*P4*P8 = 0 or A(P2) ! = 1;

Ensures that the value of point 2, point 4, and point 8 are not equal to 0. Also ensures that P2 passes the connectivity test

Condition4: P2*P4*P6 = 0 or A(P4)! =1

Ensures that points P2, P4, P6 are not set to 0
Ensures that P4 passes the connectivity test.

5. Multi-Cores Additions

Parallel Binarization:

The binarization algorithm described earlier was adapted in order to run and take advantage of the multiple cores available on both the Sun Niagara machine as well as the Lenovo T61 laptop.

Each image is divided into strips based on the number of cores available. Each strip of the image is assigned to each individual core to be processed for the binarization process.

Initial Approach: Initially each core was assigned a strip of the image to process. Once it completed the calculations all the data was aggregated in core 0 at which point the binarized image was created and broadcasted to all the nodes. Although in this case all the nodes were processing the data in unison, this approached was dropped because it was believed that though the data is indeed being calculated in parallel and does demonstrate those principles accurately, the data passing all goes to core 0 which may floods the pipe line. In order to avoid the blocked communication data line, a different approach was taken.

Second Approach: Since the communication line was blocked the next approach that was attempted was to limit the need of excessive communication. Instead of having all the cores send the data out to core 0, each even node, calculated its data, and sent it out to the next even core. For example, core 32 would calculate its data and send it to core 30, and 30 to 28 and so on. The odd cores would mirror this behavior until finally all the data from the even cores was collected in core 2 and the odd branch data in core 1. Core 2 and core 1 would transmit their branch data to core 0 where it would be used to finally combine the image. Core 0 would then broadcast the final binarized image to all the cores in order to have the same data available on all cores for the next stage.

Although the second approach solved the issue of the flooded pipeline, it brought up another issue. The data was indeed being transmitted and collected without abusing the bandwidth, but very little of the data was parallelized. In reality, Of the 8 cores and 4 threads per cpu that were utilized, only 2 cores were actually being utilized at any given time. All of the cores were blocking waiting for data before they could proceed. Hence of the 32 cores, at any one point in time, only 2 of them were operational. The Second approach solved one problem and created a new one.

Final Approach: The cores available were ordered in a binary tree like structure. All the nodes (nodes are synonymous to cores or threads in this discussion) are computing the actual data in parallel, though blocking does occur when certain nodes are waiting for data to be received. Each node will send the version of the image it modified up the hierarchy. The parent node will then take the data it received from each of its children, and combine the changes they've made. It will then add its own modification before sending its final result up the tree. Once the data reaches node 0 which is the root of the tree, the final image is then broadcasted out to all the nodes so that a fully binarized image is available to all the nodes so they can properly thin the image.

Merging the data collected proved to be quiet easy, since the program can simply look for a purely black point or a purely white point that denotes the binarized portion, and utterly ignore any other points. Although the binarization algorithm could have been corrected slightly so that less data is being sent over the pipe line, (ie. The image strips rather then the full image) since the thinning algorithm takes that said approach, this binarization was left unmodified to for data analysis purposes.

Although there was some discussion about OTSU's algorithm earlier in this document,

because of the limitation of floating point math on the Niagara processor, OTSU method was not used in this project. A more a generic approach, where a suitable mid point was chosen, and any pixel greater then that value (140) was turned white and anything less became black. OTSU method or any other binarization algorithm would have been more suitable and far more appropriate if the goal of this project was to delve into graphics and image manipulation. Unfortunately, that was not the focus of this project. The choice to omit a proper binarization algorithm was made in order to further investigate and elaborate on the parallel computing aspect of the problem.

Parallel Thinning/Skeletonization:

For the thinning process, Hilditch's Algorithm was used virtually unmodified. Unlike OTSU's Method, the thinning algorithm required little to no floating point math, therefore there was no need to modify the algorithm.

Initial Approach: The initial approach for thinning was fairly similar to the initial approach to the binarization process. Each core was assigned part of the image to thin, and run through the process. Each core once it finished its calculations, transmitted the image up to core 0, at which point core 0 would take the image and merge it into the final image, based on the source node that it received the data from. The offset for the image strip to be inserted was based on the source of the core that sent the image. The same problem occurred with this approach as the binarization where the pipe was getting flooded. We moved on to the 2nd approach in order to hopefully resolve this problem. Although parallelization of the calculations did occur, the data being sent over to core 0 was enough that it outweighed the benefit of the parallel processing.

Second Approach: The same approach was taken as the binarization process. Each even and odd core was assigned a strip of the image to thin, and it passed its data to the appropriate core down the line. The highest numbered core started processing first and moving down to the lowest core, which would be core 1 or 2, depending on if it was from an even or odd branch. The result were accumulated in core 0, then to using the calculated offset the two images were merged. The process was repeated down the line until a fully thinned image was produced at core 0. The data would then be broadcasted out to all the cores, making for a final image. The problems once again, is that the process of obtaining the image cut down on communication, but reduced the amount of parallel computing. In order to resolved this issue, a 3rd approach was attempted.

Third Approach: The nodes were ordered in a binary tree fashion, where the image was passed up the hierarchy. Each parent node would then combine the data passed from the children, add its own modifications, and push the data up to the parent node. The process would continue until the root of the tree is reached. The root node would then put the image back together, and broadcast the final result to all the cores so that they all have the same data available. Initially the approach seemed perfectly feasible but the merging of the data that was received proved to be far more complex then expected. The previous approaches relied on standard basic arithmetic calculations that could be derived based on the core which was the source of the data, or the fact that the even and odd branches would simple alternate and the size of each strip was all that was needed in order to merge that data. The program was

redesigned to attempt to tag the boundaries of the image, but that proved equally problematic when strips of the image that were back to back, the starting and ending points started to become less and less apparent and the final image did not look anything like the expected result. A modification to the third approach was used in order to develop an algorithm that would be suitable for the thinning problem.

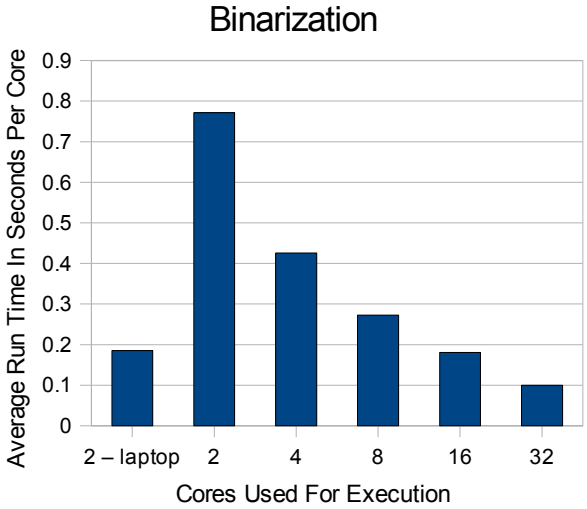
Final Approach: The final approach that was used was very similar to the third approach. A tree structure was used, but unlike the final approach of the binarization process, only a strip of the image was passed on. Each parent node did not try to merge the strips into a final image, since the merging process of unordered strips proved to be problematic. Instead, each node up the hierarchy was used as a relay point, where each node would pass along the data it processed as well as its children nodes up the hierarchy as well as an index denoting the node that has calculated that strip of data. Eventually all the data would make its way to node 0. Although the communication issue is only slightly improved, each level of the tree can be computing data concurrently without the need to affect the other nodes. Each level once it finished its calculations would then propagate the data up the tree. Unlike the third approach, only the image strip is passed up the hierarchy, not the entire image as is the case of the binarization algorithm implemented.

6. Data Analysis:

Architecture: Intel(R) Core(TM)2 Duo CPU T7500 @ 2.20GHz
UltraSPARC T1 processor (Niagara) with CoolThreads technology

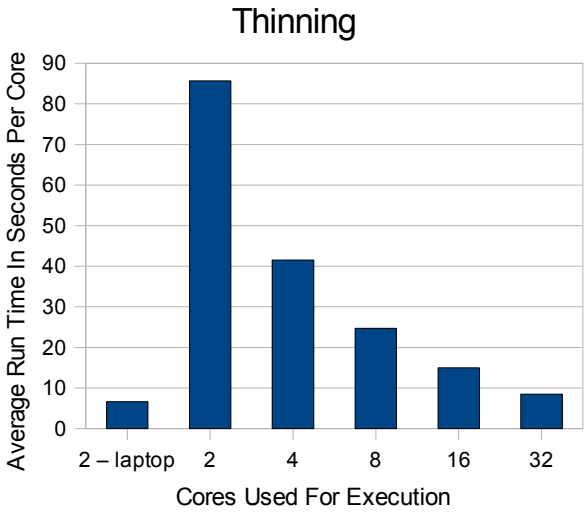
Dataset: is a collection of 49, 8-bit tiff images of 504 by 480 with file size ranging from 73k to 110k.

Binarization Run Time: (Average computation per core)



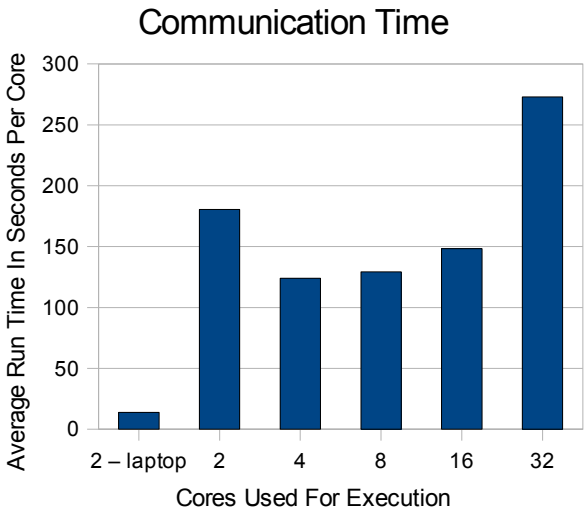
The Binarization process of an image as represented above reflects the expected trend. As the number of cores grows, the size of the strip of the image that is assigned to each core, also grows in kind. Therefore the lower the number of cores available, the longer it takes to process the data. As the number of cores increases, the time it takes to calculate each portion of the image per each core decreases as well. The run time for the laptop is so much lower mainly based on the fact that the Intel core 2 Duo is such a faster processor then the Niagara. The raw computational power per core is greater, therefore the average performance per core is also greater as reflected by the graph above.

Thinning Run Time: (Average computation per core)



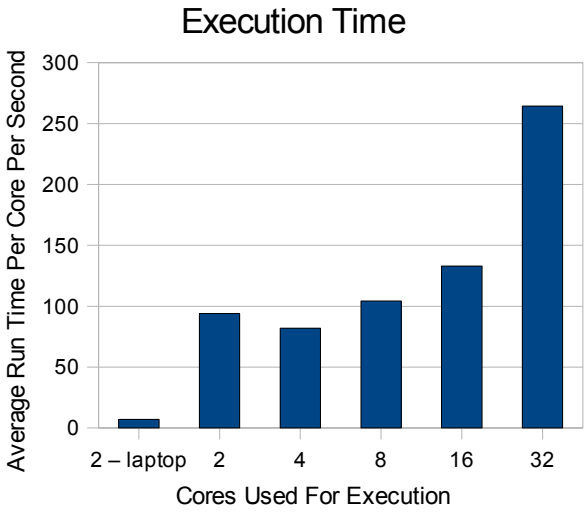
Thinning run time also reflects the same trend as it does for the binarization process. The time taken by raw calculations decreases as we add more and more cores to the problem. Since the problem gets more and more subdivided and the work that each core has to perform gets smaller and smaller. It follows that the performance based solely on raw calculations, without taking into account the communication aspect only improves with more cores.

Communication Time: (Based on run time per core minus the processing time of thinning and binarization)



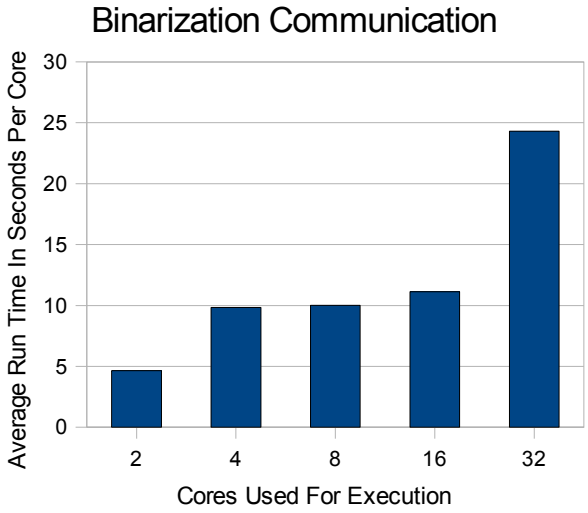
Communication time based on both the thinning and binarization algorithms is slightly elevated when 2 cores are used. It shows a drop in average communication per core once a more tree like structure is established. The cores being organized in the intended data structure format allows it to take advantage of a brief reduction in the need for extra communications which isn't as apparent when only 2 cores are utilized. Once we start adding more and more cores, though the advantage of the tree structure is diluted, and more time is spent on communicating the data back and forth then in the actual data processing.

Total Run Time Execution Time:



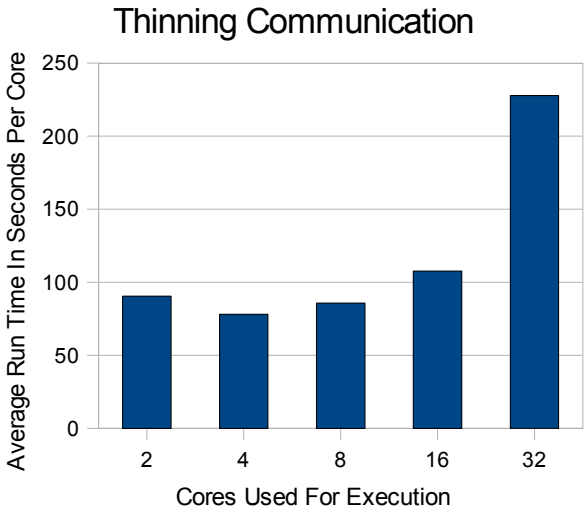
Total Execution time is simply the summation of the communication and the processing being performed by each core. As can be seen from the graph, as the tree structure takes shape, a slight improvement in performance can be seen around 4 cores. Once more and more cores are added the performance starts to deteriorate, and the 32 core test case is simply absolutely horrible. So much time is wasted in communicating that the advantage of splitting the problem up among all 32 cores is simply non-existent anymore.

Communication Time for Binarization Process



Since a different approach was taken for binarization rather then the thinning process when it comes to the communication aspect. The entire image being sent over each core rather then simply sending the actual data that would need to be compared, a separate analysis was done purely on the binarization process. As noted in the graph above, the communication cost for binarization seems to be relatively stagnant up until we hit 32 cores where the time wasted in communication seems to escalate. Again partially due to the fact that the entire image is being sent over for a more and more diminishing piece of work. The cost of sending the entire image though doesn't seem to affect the over all performance until it gets overly excessive as demonstrated in the last test case.

Communication Time for Thinning Process



The thinning process in regards to communication between cores, sends two items over, an index value and the actual strip of data that was manipulated. Unlike the binarization process where each node of the tree does a merge on the image and pushes the data up the tree in where each node does a bit of the work. The thinning process sends a small bit of the data but most of the work is performed by node 0 as far as the actually merging of the data. Each node only calculates and process each section it's allocated. At which point, the node acts as a relay point and pushes the data up the tree until all the data and index values are received at node 0 which in turn proceeds to combine the image to make the final product. The communication cost seems to be relatively stagnant as well up until the number of cores grows too high, at which point the performance deteriorates.

7. Analysis of Results and future improvements

Since so much data is being passed back and forth between each cpu core and how little actual time is being spent processing the data for the image manipulation. The benefit gained by using multiple cores rather then a single thread is starts to be less and less apparent as more and more cores are thrown to the problem. Also, using an application that is far more computationally heavy would give better results, and would most likely show an improvement as the number of cores improve. The current data seems to indicate that the ideal runtime is hit at around 4-8 cores and proceeds to deteriorate as more and more cores are added.

The program also passes data from each even core down the hierarchy until core 0 is reached, to avoid saturating the data pipe of core 0. So little time is actually spent on the actual calculations, that most of the time spent by the program is allocated to passing the image data back and forth between cores and putting the data together for the final result. Any application that aims to take advantage of parallel computing should be computationally

intensive. The cpu cost of processing a single image on a single core is so minimal, that splitting that work among n cores does not give as much of a gain in performance or speed as one would have liked. If the problem required far more cpu cycles, then the use of parallel computing would be more desired. The balance between computing and communication will always be a problem that one needs to address. One must understand the problem he is trying to solve and design both the code and the infrastructure around that limitation and design it to avoid excessive communication and allow for the gain in performance made possible by multi-core computing.

Future improvements would implement a fully functional finger print analysis program. An implementation of an image search algorithm to both analyze and match a fingerprint accurately, as well as adapt to human error and other factors which many times lead to false positives. The power of parallel computing would definitely be useful, but it seems the search problem is far easier to solve then the image manipulation issue which was tackled on for this project. Image manipulation and pattern matching would work perfectly fine if the actual image wasn't being split up among 32 cores. Assigning a subset of the image database to each core and allowing it to fully control that search space would make for an improved performance. It would allow the program to focus on the task at hand rather then focusing too much on message passing between cores. I believe that to fully take advantage of parallel computing, one needs to try to limit the need for communicating between cores unless absolutely necessary. Allowing each core to act independently would only result in an improvement in performance. Furthermore, if one looks into the actual operating system implementation of multi-cores and into the ability to force the program to run on the specified cores, an improvement in performance should be visible. Again, many of these future improvement are mere speculation, but a worthy exercise for anyone with an interest in the field.

Work Cited

1. “Overview”, April 2008; “http://www.sun.com/processors/UltraSPARC-T1/”
2. “Intel® Core™2 Duo Processor Overview”, April 2008;
3. “Download fingerprint or face algorithm demo software, SDK trials, datases.”, April 2008; http://www.neurotechnologija.com/download.html#db
4. “Finger Print Database”, January 2007; http://www.east-shore.com/data.html
5. “Otsu's Method,” April 2008; http://en.wikipedia.org/wiki/Otsu%27s_method
“http://www.intel.com/products/processor/core2duo/index.htm”
6. “OTSU's Thresholding Method,” April 2008; http://sampl.ece.ohio-state.edu/EE863/2004/ECE863-G-segclust2.ppt
7. “Skeletonization,” August 1998; http://cgm.cs.mcgill.ca/~godfried/teaching/projects97/azar/skeleton.html